NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

**A TYPE INFERENCE ALGORITHM
AND TRANSITION SEMANTICS
FOR POLYMORPHIC C**

by

Mustafa Özgen

September 1996

Thesis Advisor: Dennis Volpano

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
A Type Inference Algorithm And Transition Semantics For Polymorphic C

5. FUNDING NUMBERS

6. AUTHOR(S)
Özgen, Mustafa

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

In an attempt to bring the ML-style type inference to the C programming language, Smith and Volpano developed a type system for a dialect of C, called PolyC [SmV96a] [SmV95b]. PolyC extends C with ML-style polymorphism and a limited form of higher-order function.

Smith and Volpano proved a type soundness theorem that basically says that evaluation of a well-typed PolyC program cannot fail due to a type mismatch. The type soundness proof is based on an operational characterization of a special kind of semantic formulation called a natural semantics. This thesis presents an alternative semantic formulation, called a transition semantics, that could be used in place of the natural semantics to prove type soundness. The primary advantage of the transition semantics is that it eliminates the extra operational level, but the disadvantage is that it consists of many more evaluation rules than the natural semantics. Thus it is unclear wheter it is a suitable alternative to the two-level approach of Smith and Volpano.

Further, the thesis gives the first full type inference algorithm for the type system of PolyC. Despite implicit variable dereferencing found in PolyC, the algorithm turns out to be a rather straightforward extension of Damas and Milner's algorithm W [DaM82]. The algorithm has been implemented as an attribute grammar in Grammatech's SSL and a complete source code listing is given in the Appendix.

14. SUBJECT TERMS
programming language, polymorphism, type inference, semantics, transition semantics.

15. NUMBER OF PAGES
126

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

A TYPE INFERENCE ALGORITHM AND TRANSITION SEMANTICS FOR POLYMORPHIC C

Mustafa Özgen
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1990

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1996

Author: Mustafa Özgen
Mustafa Özgen

Approved by: Dennis Volpano
Dennis Volpano, Thesis Advisor

Craig W
Craig W. Rasmussen, Second Reader

Ted Lewis
Ted Lewis, Chairman
Department of Computer Science

ABSTRACT

In an attempt to bring the ML-style type inference to the C programming language, Smith and Volpano developed a type system for a dialect of C, called PolyC [SmV96a] [SmV96b]. PolyC extends C with ML-style polymorphism and a limited form of higher-order function.

Smith and Volpano proved a type soundness theorem that basically says that evaluation of a well-typed PolyC program cannot fail due to a type mismatch. The type soundness proof is based on an operational characterization of a special kind of semantic formulation called a natural semantics. This thesis presents an alternative semantic formulation, called a transition semantics, that could be used in place of the natural semantics to prove type soundness. The primary advantage of the transition semantics is that it eliminates the extra operational level, but the disadvantage is that it consists of many more evaluation rules than the natural semantics. Thus, it is unclear whether it is a suitable alternative to the two-level approach of Smith and Volpano.

Further, the thesis gives the first full type inference algorithm for the type system of PolyC. Despite implicit variable dereferencing found in PolyC, the algorithm turns out to be a rather straightforward extension of Damas and Milner's algorithm W for functional languages [DaM82]. The algorithm has been implemented as an attribute grammar in Grammatech's SSL and a complete source code listing is given in the Appendix.

DISCLAIMER

The computer program in the Appendix is supplied on an "as is" basis, with no warrantees of any kind. The author bears no responsibility for any consequences of using this program.

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I would like to thank Dr. Dennis Volpano for his help, advice and time during this research work. His examples of professional competency and attention to detail will help guide me in my future work.

I would also like to thank Dr. Craig Rasmussen for his assistance in editing and correcting my work.

I. INTRODUCTION

If one studies some of the well-known algorithms in Computer Science carefully, it becomes clear that some do not make any assumptions about the structure of the objects they manipulate. In other words, the algorithm can be generalized to objects of infinitely many different types. For instance, a sorting algorithm works for any type of value provided that an ordering relation can be defined for the values of the type. Also, a function, say *length,* that finds the length of a list object, is not concerned with the structure of the list elements. The result is always a natural number regardless of the type of the elements in the list. So the *length* function is *polymorphic* in the sense that it can work on infinitely many different types. What we gain from this generalizability property is that the function can have the same source code, or for that matter, the same executable(binary) for each different type of list.

An implementation of *length* in ML is given by the program below:

$$\begin{array}{l} \textbf{fun}\ \mathit{length}\ [\,] = 0 \\ \quad |\ \ \mathit{length}\ (x :: xs) = 1 + \mathit{length}\ xs\,; \end{array}$$

How can we express this polymorphic behaviour in the type of *length*? Since the type of the list elements is not relevant to the computation, we introduce a type variable to denote the type of list elements and bind it with a universal quantifier. The type of *length* is then written as

$$\forall\gamma.\gamma\ \mathit{list} \rightarrow \mathit{int}\,.$$

By instantiating the type variable γ in this type formula with different types, we can specialize the type of the function for different lists. For instance, following type formulae show two different specializations, one for integer list, and one for real list:

$$\mathit{int\ list} \rightarrow \mathit{int}$$

$$\mathit{real\ list} \rightarrow \mathit{real}$$

We contrast different forms of polymorphism in modern programming languages below.

A. MACRO-BASED POLYMORPHISM

Ada and C++ implement the idea of polymorphism in the form of Ada generics and C++ templates. In these languages, a type parameter for each of the polymorphic type variables has to be specified explicitly. Before applying an Ada generic function to a value of type τ, one has to create a specialized instance of the function for type τ explicitly in the source program. In C++, instantiation is done by the compiler vice the user; but the programmer has to provide the actual type with which the parameterized type variable will be instantiated.

The reason for the earlier specialization requirement is that, in these languages, only the same *source code* is used for a polymorphic function. But for each different type of argument, different *executable* code is generated. This kind of polymorphism is *syntactic*, since the generic instantiation is done at compile time with actual-type values that must be available at compile time. Thus, a generic procedure can be considered as an abbreviation for a set of monomorphic procedures with the same behaviour. This is called macro-based polymorphism. An alternative to macro-based polymorphism is *parametric* polymorphism, as used in Standard ML. The key difference is that polymorphic functions have an evaluation semantics. Moreover, the same executable code in addition to the same source code can be used for a polymorphic function.

B. ML-STYLE POLYMORPHISM

ML does not require programs to be annotated with types by the programmer; instead, the type of a program is inferred by the compiler without sacrificing the polymorphism. ML-style polymorphism will be discussed in the context of the Hindley/Milner system since the ML type system is based on it.

C. TYPE SYSTEMS AND TYPE SECURITY

Although we earlier assigned types to the function *length*, we did not explain how these types can be found in a systematic way since it is not always the case that programmers construct type-correct programs. In general, we prefer languages that verify the type correctness of programs statically, by checking the type correctness of every term of a program rigorously (strong typing). The main aim of strong typing is to ensure that the values are treated appropriately according to their structures, so that the evaluation of a program does not abort because of type errors. If $1 + true$ does not make sense with respect to the semantics of a language then one expects the compiler find this error before the evaluation of the program. For instance, if $+$ denotes the addition of two integer values, then at compile time it should be ensured that in an application of $+$, the parameters are terms of integer type. So we need some system of rules which tells us how to give a type to each kind of term in the language.

Such a rule system is known as a *type system* for the language. Most of the type systems are written as natural deduction systems. Below is a typical typing rule for function application:

$$\frac{A \vdash e_1 : \tau_1 \rightarrow \tau_2,\ A \vdash e_2 : \tau_1}{A \vdash e_1\, e_2 : \tau_2}$$

In this rule,

$$A \vdash e_1 : \tau_1 \rightarrow \tau_2$$

is called a *type judgement* and we say that e_1 has the type $\tau_1 \rightarrow \tau_2$ with respect to the assumption set A. Type information for the free identifiers of e_1 is taken from the assumption set A[1]. If there is no type assumption for a free identifier in the assumption set then we say e_1 is not well-typed or is ill-typed. We say that a term e

[1] When the language is extended with imperative features, A has to be extended with the assumptions about the type of memory addresses. This issue will come up in Chapter II.

is well-typed with respect to A if there is a type τ such that $A \vdash e : \tau$. An assumption set is also called a *type environment.*

In an explicitly-typed programming language, where the programs are annotated with type information, type checking ensures that type annotations are consistent with the type system. On the other hand, the types of programs including the parameterization of types can be inferred statically by the compiler without requiring any type annotations in the source code. This idea is one of the reasons for the huge success of ML, which does type inference instead.

We want programs to run without *run-time type errors.* For this reason we develop two orthogonal systems of rules, namely a *type system* and a *semantics.* If the type system types a program correctly then the evaluation of this well-typed program does not get stuck due a type error. The security from run-time type errors is known as the *soundness* of a type system. The type-soundness proof of a purely functional type system is typically more straightforward than that of an imperative type system with first-class references(pointers), first-class functions, and polymorphism. Coexistence of first-class references and polymorphism is the main source of difficulty, and it requires a precise formulation of the polymorphic treatment of references as well as a careful formulation of the semantics of a language. Damas's faulty proof of a type-soundness theorem [Dam85] is an illustration of this difficulty [Tof90].

1. Hindley/Milner Type System

Hindley's type discipline [Hin69] introduces type variables in type expressions without any quantification. Later, Milner introduced quantification of type variables [Mil78]. Damas and Milner gave an application of these ideas in a purely functional setting [DaM82]. The Hindley/Milner type system has three important properties: *parametric polymorphism*, *type inference* and *soundness and completeness of type inference.*

a. Parametric Polymorphism

The polymorphism used in Hindley/Milner system is also called *let polymorphism*, because polymorphic functions are allowed only in the local scope of a **let** construct together with a notion of *instantiation.* In

$$\textbf{let } x = e_1 \textbf{ in } e_2 \,,$$

if e_1 has the type τ with respect to A then x is assumed to have type σ, which is found by quantifying the type variables that occur in τ but do not occur free in the assumption set A. Then x binds all free occurrences of x in e_2, each of which has as its type an instance of σ.

The Hindley/Milner system imposes a restriction on the quantification: all type formulae have to be in *prenex normal form*; in other words, quantification must be done at the *outermost* level. A type formula in prenex normal form is also called a *shallow type*.

It should be noted that **let** $x = e_1$ **in** e_2 can be thought of as an abbreviation for $(\lambda x.e_2)e_1$ as far as the evaluation of these two constructs are concerned. But there is a difference between them when it comes to how they are treated by the type system. In **let** $x = e_1$ **in** e_2, e_1 can be typed polymorphically, but in $(\lambda x.e_2)e_1$, e_1 has to be monomorphic, since otherwise the type formula computed for it would not be in prenex normal form! Assume we give e_1 the type σ, which is universally quantified over some type variables, and e_2 the type τ. Then $\lambda x.e_2$ has to be given the type $\sigma \rightarrow \tau$, which is clearly not in prenex normal form.

b. Type Inference

There is an efficient algorithm, called W [DaM82], for the type system. W determines whether a given program is well-typed and infers *the most general(principal)* type for it.

Starting from the leaves of the parse tree of a program with an empty assumption set[2] , W implicitly annotates the program with type information and, at the end, either finds the *principal type* of the program, if the program is well-typed, or fails. Roughly speaking, a *principal type* is one from which all other types of the program can be derived. In the next chapter we will show, in detail, how an extension of W infers types for well-typed programs in *Polymorphic C.* Restricting the type formulae to prenex normal form allows the use of Robinson's *first order unification algorithm* [Rob65].

c. Soundness and Completeness

In [DaM82] it is shown that W is sound, in the sense that it finds types only for well-typed expressions, and complete, in the sense that if a program is a well-typed then W finds the most general type for it.

[2] Actually, a type assignment process never starts with an empty assumption set if there are built-in operations in the language but we would like to consider the emptiness of the assumption set in terms of adding a new assumption to the set during the process of type assignment.

II. THE POLYMORPHIC C LANGUAGE

This section gives an overview of *Polymorphic C*. Hereafter we use *PolyC* instead of *Polymorphic C* as a shorthand. The reader should see [SmV96a] for a detailed account of PolyC.

PolyC is designed to incorporate an advanced polymorphic type system, similiar to those designed as extensions to the core-ML type system, into the widely used imperative programming language, C. Unlike other extensions, the PolyC type system also captures polymorphic typing of first class pointers.

PolyC is semantically very close to K&R C [KR78], with the same pointer operations, including the address of &, the dereferencing *, and pointer arithmetic. The main design rationale was to bring ML-style polymorphism and type security to C while keeping the flexibility and simplicity of C. Variables in PolyC are second class and implicitly derefenced, while pointers are first class and explicitly dereferenced by the * operator.

As a new feature, functions are first class citizens in PolyC, and, as in C, function applications are implemented on a stack without use of static links or displays by imposing a restriction on functions: *The free identifiers of a function must be declared at top level; that is, the scope of the declaration must extend all the way to the end of the program*[SmVo95]. In C, no automatic variable[1] can occur free in a function declaration so that a function declaration is closed with respect to the top-level (global) identifier set. PolyC establishes the same property via this restriction by ensuring that a lambda-bound identifier, or an identifer bound by a **let**, **letvar** or **letarr** declaration whose scope does not extend to the end of the program, does not occur free in a function. In the program below, the scope of y does not extend to the

[1] A variable that is created as a result of a function application. In other words, the local variables of a function including its formal parameters.

end of the program, so $\lambda z.z{+}y$ is not closed with respect to top-level identifiers.

$$\begin{aligned}&\textbf{letvar } x := \textbf{letvar } y := 5 \textbf{ in } \lambda z.z + y\\ &\textbf{in } \ldots\end{aligned}$$

But this restriction has another consequence: Currying of functions is not allowed anymore. An attribute grammar enforcing the restriction is given in Appendix.

PolyC does not distinguish between *commands* and *expressions*. Every term of the language is an expression. A subset of expressions, however, are distinguished as *Values*, which are the syntactic values[2] of the language. The core syntax is given below.

$$\begin{array}{lll}
(\mathit{Expr}) & e ::= & v \mid e(e_1,\ldots,e_n) \mid e_1 := e_2 \mid \\
& & \&e \mid *e \mid e_1 + e_2 \mid e_1[e_2] \mid e_1;e_2 \mid \\
& & \textbf{while } e_1 \textbf{ do } e_2 \mid \\
& & \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \mid \\
& & \textbf{let } x = e_1 \textbf{ in } e_2 \mid \\
& & \textbf{letvar } x := e_1 \textbf{ in } e_2 \mid \\
& & \textbf{letarr } x[e_1] \textbf{ in } e_2 \mid \\
& & (a,1) \\
(\mathit{Values}) & v ::= & x \mid c \mid \lambda x_1,\ldots,x_n.e \mid (a,0)
\end{array}$$

Meta-variable x ranges over identifiers, c over literals (such as integer literals and **unit**), and a over addresses. To be able to catch pointer errors in the semantics, an address is designed as a pair (i,j), where i is a *segment* and j is an *offset* in that segment. The lifetime of a cell ends when the scope of the identifier to which it is bound ends.

Since core PolyC does not support overloading, + denotes only pointer arithmetic and $*$ denotes dereferencing. The construct **letvar** binds x to a new cell

[2] Syntactic values correspond to *non-expansive* expressions of [Tofte90], where evaluation of a non-expansive expression does not extend the domain of the store function.

initialized to value of e_1; the scope of binding is e_2 and the lifetime of the cell ends after the evalution of e_2. If e_1 has type τ then x has type $\tau\ var$. Analogously, the construct **letarr** binds x to a pointer to the first cell of n consecutive uninitialized cells where n is a positive integer found by the evaluation of e_1; the scope of x is e_2, and the lifetime of the array ends after e_2 is evaluated.

Having functions as first class citizens leads to a more flexible syntax than that of C. In addition to named functions, users can define anonymous functions easily anywhere in the program such as

$$\textbf{let } id = \lambda x.x \textbf{ in } id(\lambda y.y+1).$$

PolyC does not have an explicit syntax to create uninitialized identifiers of pointer type. But it unifies array types and pointers, as in C. Then declaring an array of size 1 is the declaration of an uninitialized pointer type identifier.

Another subtle syntactic difference is in the treatment of the formal parameters of a function. In C, formal parameters are considered as local variables of a function, whereas they are treated as constants in PolyC. But it is not hard to achieve a C-like treatment by declaring new local variables in the body of the function and initializing them to the values of the formal parameters. Below, a C function and its PolyC version are given in order.

$$int\ f(int\ x)\{\ldots return\ x;\}$$

$$\textbf{let } f = \lambda x.\text{letvar}\, x := x \textbf{ in } x \textbf{ in } \ldots$$

A. THE TYPE SYSTEM

ML stratifies the types into two levels: the ordinary $\tau - types$ ($data\ types$) and $\sigma - types$ ($type\ schemes$). PolyC adds another level to this stratification, namely $\rho - types$ ($phrase\ types$) to establish the second-class status of variables. Types of PolyC are given below [SmV96a] :

$$\begin{array}{llll} \tau & ::= & \alpha \mid \mathit{int} \mid \mathit{unit} \mid \tau\ \mathit{ptr} \mid \tau_1 \times \cdots \times \tau_n \to \tau & (\textit{data types}) \\ \sigma & ::= & \forall\alpha\,.\,\sigma \mid \tau & (\textit{type schemes}) \\ \rho & ::= & \sigma \mid \tau\ \mathit{var} & (\textit{phrase types}) \end{array}$$

Meta variable α ranges over type variables.

The type system is designed as a natural deduction system to assign types to expressions. It is given in Figure 1 [SmV96a].

In Section B, we saw that the type of a term is found with respect to an assumption set A, where A ranges over identifiers and assigns types to free identifiers of a term. Having A range over identifiers only is adequate for sound typing in a functional setting, but if the language includes assignable locations, then we have to be able to implicitly type a location, regarding the value stored in it, to get a handle on the soundness of the type system. Intuitively, a location must be given a monotype since we can not store different types of values in a location. A thorough discussion of the difficulties with references in a polymorphic type system is given in [Tof90]. As given in Figure 1, typing judgements have the form

$$\lambda; \gamma \vdash e : \rho,$$

meaning that expression e has type ρ, assuming that γ prescribes phrase types for the free identifiers of e and λ prescribes data types for the variables and pointers in e. More precisely, meta-variable γ ranges over *identifier typings*, which are finite functions mapping identifiers to phrase types; $\gamma(x)$ is the phrase type assigned to x by γ and $\gamma[x : \rho]$ is a modified identifier typing that assigns phrase type ρ to x and assigns phrase type $\gamma(x')$ to any identifier x' other than x. Similiar conventions apply to $\lambda(x)$ and $\lambda[x : \rho]$ [SmV96a].

Generalization of data type τ with respect to λ and γ is denoted by $\mathit{Close}_{\lambda;\gamma}(\tau)$ and is equivalent to the type scheme $\forall\bar{\alpha}\,.\,\tau$, where $\bar{\alpha}$ is the set of all type variables occurring free in τ but not in λ or in γ. We write $\lambda \vdash e : \tau$ and $\mathit{Close}_{\lambda}(\tau)$ when $\gamma = \emptyset$.

(VAR-ID) $\lambda; \gamma \vdash x : \tau\ \mathit{var} \qquad \gamma(x) = \tau\ \mathit{var}$

(IDENT) $\lambda; \gamma \vdash x : \tau \qquad \gamma(x) \geq \tau$

(PTR) $\lambda; \gamma \vdash ((i,j),0) : \tau\ \mathit{ptr} \qquad \lambda(i) = \tau$

(VAR) $\lambda; \gamma \vdash ((i,j),1) : \tau\ \mathit{var} \qquad \lambda(i) = \tau$

(LIT) $\lambda; \gamma \vdash c : \mathit{int} \qquad c$ is an integer literal

$\lambda; \gamma \vdash \mathbf{unit} : \mathit{unit}$

($\rightarrow$-INTRO)

$$\frac{\lambda; \gamma[x_1 : \tau_1, \ldots, x_n : \tau_n] \vdash e : \tau}{\lambda; \gamma \vdash \lambda x_1, \ldots, x_n . e : \tau_1 \times \cdots \times \tau_n \rightarrow \tau}$$

($\rightarrow$-ELIM)

$$\frac{\begin{array}{l}\lambda; \gamma \vdash e : \tau_1 \times \cdots \times \tau_n \rightarrow \tau, \\ \lambda; \gamma \vdash e_i : \tau_i, \ \ 1 \leq i \leq n\end{array}}{\lambda; \gamma \vdash e(e_1, \ldots, e_n) : \tau}$$

(LET-VAL)

$$\frac{\lambda; \gamma \vdash v : \tau_1, \ \ \lambda; \gamma[x : \mathit{Close}_{\lambda;\gamma}(\tau_1)] \vdash e : \tau_2}{\lambda; \gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ e : \tau_2}$$

(LET-ORD)

$$\frac{\lambda; \gamma \vdash e_1 : \tau_1, \ \ \lambda; \gamma[x : \tau_1] \vdash e_2 : \tau_2}{\lambda; \gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

(LETVAR)

$$\frac{\lambda; \gamma \vdash e_1 : \tau_1, \ \ \lambda; \gamma[x : \tau_1\ \mathit{var}] \vdash e_2 : \tau_2}{\lambda; \gamma \vdash \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 : \tau_2}$$

(LETARR)

$$\frac{\lambda; \gamma \vdash e_1 : \mathit{int}, \ \ \lambda; \gamma[x : \tau_1\ \mathit{ptr}] \vdash e_2 : \tau_2}{\lambda; \gamma \vdash \mathbf{letarr}\ x[e_1]\ \mathbf{in}\ e_2 : \tau_2}$$

(R-VAL)

$$\frac{\lambda; \gamma \vdash e : \tau\ \mathit{var}}{\lambda; \gamma \vdash e : \tau}$$

(L-VAL)

$$\frac{\lambda; \gamma \vdash e : \tau\ \mathit{ptr}}{\lambda; \gamma \vdash *e : \tau\ \mathit{var}}$$

(ADDRESS)

$$\frac{\lambda; \gamma \vdash e : \tau\ \mathit{var}}{\lambda; \gamma \vdash \&e : \tau\ \mathit{ptr}}$$

(ASSIGN)

$$\frac{\lambda; \gamma \vdash e_1 : \tau\ \mathit{var}, \ \ \lambda; \gamma \vdash e_2 : \tau}{\lambda; \gamma \vdash e_1 := e_2 : \tau}$$

Figure 1. Rules of the Type System, continued next page

$$\text{(ARITH)} \qquad \frac{\lambda;\gamma \vdash e_1 : \tau\ ptr, \quad \lambda;\gamma \vdash e_2 : int}{\lambda;\gamma \vdash e_1 + e_2 : \tau\ ptr}$$

$$\text{(SUBSCRIPT)} \qquad \frac{\lambda;\gamma \vdash e_1 : \tau\ ptr, \quad \lambda;\gamma \vdash e_2 : int}{\lambda;\gamma \vdash e_1[e_2] : \tau\ var}$$

$$\text{(WHILE)} \qquad \frac{\lambda;\gamma \vdash e_1 : int, \quad \lambda;\gamma \vdash e_2 : \tau}{\lambda;\gamma \vdash \mathbf{while}\ e_1\ \mathbf{do}\ e_2 : unit}$$

$$\text{(COMPOSE)} \qquad \frac{\lambda;\gamma \vdash e_1 : \tau_1 \quad \lambda;\gamma \vdash e_2 : \tau_2}{\lambda;\gamma \vdash e_1 ; e_2 : \tau_2}$$

Figure 2. Rules of the Type System, cont.

Typing a **let** construct is done via two rules, namely LET-VAL and LET-ORD. If e_1 is a syntactic value then LET-VAL is used and x is given a phrase type by genaralizing the type of e_1. On the other hand, LET-ORD is defined for the cases where e_1 is not a syntactic value and no type generalization is allowed. Regarding these two rules, all of the type variables in PolyC can be seen as imperative(weak) when compared to Standard ML type system [Tof90].

1. Examples of Type Inference

Consider the program

$$\mathbf{let}\ id = \lambda x.x\ \mathbf{in}\ id(\lambda y.y+1)\,;id(3)\ .$$

We start with empty domains for λ and γ. The LET-VAL typing rule is the first one to start with since $\lambda x.x$ is a value. By the first premise of LET-VAL, $\lambda x.x$ is given the type $\alpha \to \alpha$. We extend γ with $x : \forall\alpha.\alpha \to \alpha$ by closing $\alpha \to \alpha$ with respect to λ and γ, and try to type the sequence $id(\lambda y.y+1);id(3)$. The first expression of the sequence is typed using $\to$-ELIM. We instantiate id as $\beta \to \beta$ and $\lambda y.y+1$ is given the type $\zeta\ ptr \to \zeta\ ptr$. Rule $\to$-ELIM requires β and $\zeta\ ptr$ be the same, so we unify them with representative type $\zeta\ ptr$. The second expression is also typed by $\to$-ELIM. We instantiate id to $\xi \to \xi$ this time, and 3 has type int. By $\to$-ELIM, ξ

and *int* are unified to *int*. So the result of the application has type *int*. Then by COMPOSE, $id(\lambda y.y+1); id(3)$ is given the type *int*. Since, the hypotheses of LET-VAL are satisfied, it is deduced that the program has the type *int*.

The program below shows how the type system prevents memory locations from being treated polymorphically.

$$\textbf{letvar}\ id := \lambda x.x\ \textbf{in}\ id := \lambda y.y + 1\,;\ \textbf{let}\ id' = id\ \textbf{in}\ id'(3)$$

We start with the LETVAR typing rule and give the type $\alpha \rightarrow \alpha$ to $\lambda x.x$. Then we extend γ with $id : (\alpha \rightarrow \alpha)\ var$ and try to type the body of letvar, which is a sequence. The first expression of the sequence is typed using ASSIGN. The type $(\alpha \rightarrow \alpha)\ var$ is given to id by γ, and $\lambda y.y + 1$ is given the type $\beta\ ptr \rightarrow \beta\ ptr$. By ASSIGN, $\alpha \rightarrow \alpha$ and $\beta\ ptr \rightarrow \beta\ ptr$ must be the same. So we unify α and $\beta\ ptr$ with representative type $\beta\ ptr$. Finally, the assignment is given the type $\beta\ ptr \rightarrow \beta\ ptr$ and γ gives the type $(\beta\ ptr \rightarrow \beta\ ptr)\ var$ to id from now on.

The second expression of the sequence is a *let* expression. Since id is an identifier we use the LET-VAL typing rule. The type $(\beta\ ptr \rightarrow \beta\ ptr)\ var$ is given to id by γ. Since id is in an r-value context, we use rule R-VAL and find the type $\beta\ ptr \rightarrow \beta\ ptr$ for id. Then we extend γ with $id' : Close_{\lambda;\gamma}(\beta\ ptr \rightarrow \beta\ ptr)$. β occurs free in γ by the fact that it occurs free in the type judgement $id : (\beta\ ptr \rightarrow \beta\ ptr)\ var$, so $Close_{\lambda;\gamma}(\beta\ ptr \rightarrow \beta\ ptr) = \beta\ ptr \rightarrow \beta\ ptr$. Now, we try to type the body of the *let* expression which is the application $id'(3)$. The type $\beta\ ptr \rightarrow \beta\ ptr$ is given to id' by γ and 3 has the type *int*. But then $\rightarrow$-ELIM requires $\beta\ ptr$ and *int* be the same which is not possible. So we conclude that this application is not typable and therefore the program is untypable.

Having first class pointers in the language can lead to the occurrence of dangling pointers. To preserve the flexibility and expressiveness of C, PolyC does not prevent the dangling pointers but the semantics catches the dereferencing of a dangling pointer. The program below shows how a reference location escapes from its scope by returning the address of the variable y in the body of the inner *letvar* expression, and how the type system assigns a type to this program.

$$\textbf{letvar}\, x := \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&y\ \textbf{in}\ (*x)(3)$$

We start with the LETVAR typing rule to type the program. The first premise of LETVAR requires us to type the inner *letvar* expression, **letvar** $y := \lambda z.z$ **in** $\&y$. By a second use of LETVAR, we give the type $\alpha \rightarrow \alpha$ to $\lambda z.z$, and then by extending γ with $y : (\alpha \rightarrow \alpha)\ var$, the body of inner *letvar*, $\&y$, is given the type $(\alpha \rightarrow \alpha)\ ptr$. So it is deduced that the inner *letvar* has the type $(\alpha \rightarrow \alpha)\ ptr$. Now γ is extended with $x : ((\alpha \rightarrow \alpha)\ ptr)\ var$, and we try to type the body of the outer *letvar*. Since it is an application, we use →-ELIM. We type $*x$ by first using R-VAL, then L-VAL followed by R-VAL again giving type $\alpha \rightarrow \alpha$. Since 3 has the type *int*, we deduce the type *int* for the application and also for the program itself.

In Chapter III we will show how the semantics prevents the evaluation of this program by catching the dereferencing of the dangling pointer stored in x.

III. THE TYPE-INFERENCE ALGORITHM

In this chapter we present the type-inference algorithm W_c. It is similiar to Milner's algorithm W [DaM82], which is based on unification of type expressions. We also present an example type inference produced by the computer implementation of W_c. We first give some definitions about substitution and unification.

A. SUBSTITUTION AND UNIFICATION

A *substitution* S is a finite set of the form

$$[\tau_1/\alpha_1, \tau_2/\alpha_2, \ldots, \tau_n/\alpha_n]$$

where the variables α_i $(1 \leq i \leq n)$ are distinct. $S\rho$ is called the *application* of substitution S to type expression ρ. The result of $S\rho$ is another type expression ρ', obtained from ρ by replacing simultaneously each free occurrence of the variable α_i, $1 \leq i \leq n$ in ρ by τ_i , renaming the bound variables of ρ if necessary. ρ' is called an instance of ρ. Note that ρ and ρ' can be the same if no α_i occurs in τ.

We often write $S_2(S_1\rho)$ or simply $S_2S_1\rho$ for the application of the composition $S_1 \circ S_2$ to ρ. An *empty substitution* is written as [].

A substitution S is called a *unifier* for type expressions ρ_1 and ρ_2 if $S\rho_1 = S\rho_2$. We say ρ_1 and ρ_2 are *unifiable* if there is a *unifier* for them.

A unifier S is called *the most general unifier* of ρ_1 and ρ_2 if for every other unifier S' of ρ_1 and ρ_2 there is a substitution S'' such that

$$S' = S \circ S''.$$

Unification of type expressions is implemented using Robinson's first order unification algorithm, which returns a substitution U, where U is *the most general unifier* of a pair of type expressions ρ_1 and ρ_2 given as the arguments to the algorithm [Rob65]; if ρ_1 and ρ_2 are not unifiable then the algorithm fails to return such a substitution.

B. ALGORITHM W_C

W_c takes two input arguments, γ and e, and returns a pair (S, τ). As defined for the type system, γ is a finite function mapping identifiers to phrase types. The second input argument e is the expression whose type is to be inferred, S is a substitution and τ is the type inferred for e by W_c. The type returned by W_c is a $\tau - type$ in that it is called only in r-value contexts. Since locations do not occur in user programs, we do not use a location typing λ in W_c. Only γ is needed to do type inference.

$W(\gamma, e)$ is defined by cases:

```
1. $e$ is $x$
     case $\gamma(x) = \forall\alpha_1, \ldots, \alpha_n.\tau$
       return $([\,], [\beta_i/\alpha_i]\tau)$ where $\beta_i$ is new for each $1 \leq i \leq n$
     case $\gamma(x) = \tau$
       return $([\,], \tau)$
     case $\gamma(x) = \tau\ var$
       return $([\,], \tau)$.

2. $e$ is $\lambda x_1, \ldots, x_n.e_1$
     let $(S_1, \tau_1) = W(\gamma[x_1 : \beta_1, \ldots, x_n : \beta_n], e_1)$ where $\beta_i$'s are new
     return $(S_1, S_1(\beta_1 \times \cdots \times \beta_n) \to \tau_1)$.

3. $e$ is $e'(e_1, \ldots, e_n)$ then
     let $(S', \tau') = W(\gamma, e')$
     let $(S_1, \tau_1) = W(S'\gamma, e_1)$
       $\vdots$
     let $(S_n, \tau_n) = W(S_{n-1}S_{n-2} \cdots S_1S'\gamma, e_n)$
     let $S'' = Unify(C_0\tau', (C_1\tau_1 \times C_2\tau_2 \times \cdots \times C_{n-1}\tau_{n-2} \times S_n\tau_{n-1} \times \tau_n) \to \beta)$
       where $\beta$ is new,
      $C_i = S_n S_{n-1} \cdots S_{i+1}$ and $1 \leq i < n$
    return $(S_n S_{n-1} \cdots S_1 S', S''\beta)$.

4. $e$ is **let** $x = e_1$ **in** $e_2$
    let $(S_1, \tau_1) = W(\gamma, e_1)$
    if $e_1$ is a syntactic value then
      let $(S_2, \tau_2) = W(S_1\gamma[x : \mathit{Close}_{S_1\gamma}(\tau_1)], e_2)$
    else
      let $(S_2, \tau_2) = W(S_1\gamma[x : \tau_1], e_2)$
    return $(S_2 S_1, \tau_2)$.

5. $e$ is **letvar** $x := e_1$ **in** $e_2$
    let $(S_1, \tau_1) = W(\gamma, e_1)$
    let $(S_2, \tau_2) = W(S_1\gamma[x : \tau_1\ \mathit{var}], e_2)$
    return $(S_2 S_1, \tau_2)$.

6. $e$ is **letarr** $x[e_1]$ **in** $e_2$ then
    let $(S_1, \tau_1) = W(\gamma, e_1)$
    let $S' = \mathit{Unify}(\tau_1, \mathit{int})$
    let $(S_2, \tau_2) = W(S'S_1\gamma[x : \beta\ \mathit{ptr}], e_2)$ where $\beta$ is new
    return $(S_2 S' S_1, \tau_2)$.

7. $e$ is $*e_1$ then
    let $(S_1, \tau_1) = W(\gamma, e_1)$
    let $S' = \mathit{Unify}(\tau_1, \beta\ \mathit{ptr})$
        where $\beta$ is new
    return $(S'S_1, S'\beta)$.
```

```
8. $e$ is $\&e_1$ then
     case $e_1$ is $x$
       if $\gamma(x) = \tau_1\, var$ then
         return $([\,], \tau_1\, ptr)$
       else fail
     case $e_1$ is $*e_2$
       let $(S_1, \tau_1) = W(\gamma, e_2)$
       let $S' = Unify(\tau_1, \beta\, ptr)$ where $\beta$ is new
       return $(S'S_1, S'\beta\, ptr)$.
9. $e$ is $e_1 := e_2$ then
     case $e_1$ is $x$
       if $\gamma(x) = \tau\ var$ then
         let $(S_1, \tau_1) = W(\gamma, e_2)$
         let $S' = Unify(\tau_1, S_1\tau)$
         return $(S'S_1, S'\tau_1)$.
       else fail
     case $e_1$ is $*e'$
       let $(S_1, \tau_1) = W(\gamma, e')$
       let $S' = Unify(\tau_1, \beta\, ptr)$ where $\beta$ is new
       let $(S_2, \tau_2) = W(S'S_1\gamma, e_2)$
       let $S'' = Unify(\tau_2, S_2S'\beta)$
       return $(S''S_2S'S_1, S''\tau_2)$
10. $e$ is $e_1 + e_2$ then
     let $(S_1, \tau_1) = W(\gamma, e_1)$
     let $S' = Unify(\tau_1, \beta\, ptr)$ where $\beta$ is new
```

```
    let $(S_2, \tau_2) = W(S'S_1\gamma, e_2)$
    let $S'' = \mathit{Unify}(\tau_2, \mathit{int})$
    return $(S''S_2S'S_1, S''S_2S'\beta\,\mathit{ptr})$
11. $e$ is $e_1; e_2$ then
    let $(S_1, \tau_1) = W(\gamma, e_1)$
    let $(S_2, \tau_2) = W(S_1\gamma, e_2)$
    return $(S_2S_1, \tau_2)$
12. $e$ is **while** $e_1$ **do** $e_2$ then
    let $(S_1, \tau_1) = W(\gamma, e_1)$
    let $S' = \mathit{Unify}(\tau_1, \mathit{int})$
    let $(S_2, \tau_2) = W(S'S_1\gamma, e_2)$
    return $(S_2S'S_1, \mathit{unit})$
```

Function *Unify* is the implementation of Robinson's unification algorithm and $Close_{S_1\gamma}(\tau_1)$ in case 4 is the generalization of τ_1 with respect to the environment found after the application of the substitution S_1 to γ.

C_i in case 3 denotes the composition of substitutions that is applied to the type of the ith actual parameter of a function application, where $1 \leq i < n$ and n is the number of formal parameters. C_0 is the substitution composition applied to the called function.

We omit the *default* arm of case statements for simplicity and it corresponds to a *fail* case of W_c. In addition to the explicitly stated fail cases, W_c also fails if *Unify* fails to return a substitution or any subinvocation of W_c fails.

Array subscripting $e_1[e_2]$ is a syntactic sugar for $*(e_1 + e_2)$ so that we do not consider array subscripting as a separate case in W_c.

The algorithm does not explicitly specify how a "new"type variable is obtained. We assume that there is a global list of used variables, and that new ones are selected from those not in that list.

1. Sample Type Inference with W_c

An interpreter for PolyC has been written using The Synthesizer Generator environment [Gram]. It includes an implementation of W_c and the syntax and the natural semantics of PolyC given in [SmV96a] with some modifications. Source code for the interpreter is given in Appendix.

Below is an implementation of a HeapSort algorithm in PolyC [Cor90]. The type annotations shown as

$$\mathtt{id} : \sigma$$

for selected identifiers only are done automatically by the interpreter.

```
let $Swap : \forall * 9.(*9\,\mathtt{ptr} \times *9\,\mathtt{ptr} \to *9)$ $:= \lambda(a,\ b)$ {let $temp = !a$ in
                    $!a\ := !b;$
                    $!b\ := !b;$
                    end } in
    letvar $heapSize : \mathtt{int\,var} := 0$
        let $Heapify : \forall * 21.(*21\mathtt{ptr} \times \mathtt{int} \times (*21 \times *21 \to \mathtt{int}) \to \mathtt{unit})$
        $= \lambda(a,\ i,\ comp)${letvar $left : \mathtt{int\,var} := 2 * i + 1$ in
                              letvar $current : \mathtt{int\,var} := i$ in
                                  while $left \leq heapSize - 1$ do
                                      if $left \leq heapSize - 1$
                                          then if $comp(a[left], a[left + 1])$
                                              then $largest\ := left$
                                              else $largest\ := left + 1$
```

```
              fi
              else $largest := left$
            fi;
            if $comp(a[largest], a[current])$ then
              $Swap(\&a[largest], \&a[current])$;
              $current := largest$;
              $left := 2 * current + 1$
            else $left := heapSize + 1$
            fi
          od
        end
      end
    end } in
let $BuildHeap : \forall * 29.(*29\,\mathtt{ptr} \times \mathtt{int} \times (*29 \times *29 \rightarrow \mathtt{int}) \rightarrow \mathtt{unit})$
$= \lambda(a, size, comp)\{heapSize := size$;
    letvar $i := size/2 - 1$ in
      while $i \geq 0$ do
        $Heapify(a, i, comp)$;
        $i := i - 1$
      od
    end } in
  let $HeapSort : \forall * 35.(*35\,\mathtt{ptr} \times \mathtt{int} \times (*35 \times *35 \rightarrow \mathtt{int}) \rightarrow \mathtt{unit})$
  $= \lambda(a, size, comp)\{BuildHeap(a, size, comp)$;
      letvar $i := size - 1$ in
        while $i \geq 1$ do
```

```
                Swap(&a[i], &a[0]);
                heapSize := heapSize − 1;
                Heapify(a, 0, comp);
                i := i − 1
              od
            end } in
          letarr a[8] in
            a[0] := 12;
            a[0] := 5;
            a[0] := 23;
            a[0] := 8;
            a[0] := 1;
            a[0] := 45;
            a[0] := 17;
            a[0] := 51;
            HeapSort(a, 8, λ(a, b) {a > b});
            (a[0], (a[1], (a[2], (a[3], (a[4], (a[5], (a[6], a[7])))))))
          end
        end
      end
    end
  end
end
val (1, (5, (8, (12, (17, (23, (45, 51)))))))
: (int × (int × (int × (int × (int × (int × (int × int)))))))
```

In type expressions, Cartesian product × binds tighter than arrow $\rightarrow$; $*i$, where $i \in$ *Natural*, is a type variable generated by a global new type variable generator function. The second line from the last shows the result of the evaluation of the program and the last line shows the type of the program. Here we use $*$ to denote integer multiplication vice dereferencing, which is denoted by !, and $+$ denotes integer addition vice pointer arithmetic, which is denoted by $\oplus$. Type quantification is denoted by $\forall$ as in the type of *Swap*.

2. Correctness Criteria for W_c

Due to time constraints on preparation of this thesis, we are not able to pose theorems related to correctness of W_c and prove them. Roughly speaking, correctness of W_c should be established by showing that W_c is *sound* (syntactically) and *complete*. By soundness, we mean that if W_c succeeds in finding a type for a PolyC expression then that type can be derived for the expression in the type system. By completeness, we mean that if an expression of PolyC has a type at all then W_c will succeed in finding a type for this expression which is at least as general.

IV. TRANSITION SEMANTICS FOR POLYC

In this chapter we develop a transition semantics (TS) for PolyC that captures each single step of the evaluation of an expression. First we will look at the motivations behind this type of semantics.

A. STRUCTURAL OPERATIONAL SEMANTICS OF POLYC

To show the semantic soundness of the type system of PolyC, Smith and Volpano use the framework of Harper [Har94] and develop the subject reduction property using the Structural Operational Semantics(SOS) given in the same paper [SmV96a]. But the subject reduction property based on SOS does not expose enough information about the course of evaluation of a program, making it difficult to establish a semantic soundess result for the type system. SOS defines a relation between the expressions and their normal forms but does not explicitly keep track of step-by-step construction of the evaluation tree of an expression. Instead, by using the compositionality property in a coarse-grained sense, it assumes that in one or more steps the evaluation trees created by the subexpressions will constitute the final evaluation tree of an expression. If a subexpression fails to evaluate to a value, so does the whole expression. But we cannot know exactly how the subexpression got stuck, which is a key issue in being able to reason about the semantics and its interaction with the type system. SOS admits structural induction on evaluation derivations.

Gunter [Gun92] strengthens subject reduction for the pure functional programming language PCF by augmenting the evaluation rules with new rules that evaluate to a special value, namely **tyerr** which does not have a type. These rules cover the evaluation of possible ill-typed expressions. Since a well-typed expression never contains an ill-typed subexpression, then any of the rule instances that occur in the evaluation of a well-typed expression cannot be an instance of one of these new rules. Hence, it is not the case that a well-typed expression evaluates to **tyerr**. So by showing that subject reduction holds for the augmented evaluation rules, absence of run-time type errors is guaranteed. In addition to the drawback of augmenting the evaluation rules, this approach does not give us any information about the nature of the other errors that can occur during evaluation of well-typed programs, which will be an important issue in an imperative setting with assignable locations and first class pointers.

On the other hand, Smith and Volpano use the combination of subject reduction and a lemma, namely the *Correct Form Lemma* to prove a soundness theorem [SmV96a]. The *Correct Form Lemma* shows the correct syntactic form of a value when its type is given. It basically shows the type system is not being silly by giving some unexpected type to a term. For example, if a value has type $\tau_1 \rightarrow \tau_2$ then the value is a $\lambda - abstraction$ and not, say, an integer. Also, to get a handle on the "progress"of an attempted evaluation, the evaluation rules are re-cast as an instance of a recursive function, *eval*. The Soundness Theorem then shows that if an activation of *eval* aborts, it is due to one of the following four errors [SmV96a]:

E1. An attempt to read or write to a dead address (i, j).

E2. An attempt to read or write to a nonexistent address (i, j). Address $(i, 0)$ always will exist, so the problem is that the offset j is invalid.

E3. An attempt to read an uninitialized address (i, j).

E4. An attempt to declare an array of size less than or equal to 0.

But re-casting the evaluation rules as an instance of *eval* and proving a soundness result based on the abort conditions of *eval* seems a little bit informal. What we would like to do is to collect more information about the "course"of the evaluation of the programs so that we can use more formal techniques to prove a soundness result for PolyC. It is for this reason that we explore a transition semantics for PolyC.

B. TRANSITION SEMANTICS FOR POLYC

1. Definitions

First, we give some definitions used in the transition (evaluation) rules.

A *configuration* is a triple (e, μ, δ) where e is an expression, δ is an *active cell indicator*, and μ is a memory which is a finite function from addresses to values; μ may also map addresses to **dead** or **uninit**, indicating that the cell with that address has been deallocated or is uninitialized. The contents of an address $a \in dom(\mu)$ is the value $\mu(a)$, and we write $\mu[a := v]$ for the memory that assigns value v to address a, and value $\mu(a')$ to an address $a' \neq a$; $\mu[a := v]$ is an update of μ if $a \in dom(\mu)$ and an extension of μ if $a \notin dom(\mu)$.

An *active cell* is an address whose value is not **dead**. The natural number δ denotes the number of active cells created so far by an expression or by its subexpressions. We use δ for the purpose of keeping track of the lifetime of memory cells that are allocated via **letvar** and **letarr** declarations.

We define a binary relation $\rightarrow$ from configurations to configurations to capture the single step transitions. If evaluating the closed expression e in memory μ with respect to δ results in a new expression e', a new memory μ' and a new active cell indicator δ', then

$$(e, \mu, \delta) \rightarrow (e', \mu', \delta').$$

We write $[e'/x]e$ to denote the capture-avoiding substitution of e' for all free occurrences of x in e and the result of the substitution is another expression of PolyC.

2. The Transition Rules

The transition rules are given below:

(CONTENTS)

(I) $$\frac{a \in dom(\mu) \text{ and } \mu(a) = v}{((a,1), \mu, \delta) \rightarrow (v, \mu, \delta)}$$

(DEREF)

(I) $$\frac{a \in dom(\mu) \text{ and } \mu(a) = v}{(*(a,0),\mu,\delta) \rightarrow (v,\mu,\delta)}$$

(II) $$\frac{(e,\mu,\delta) \rightarrow (e',\mu',\delta')}{(*e,\mu,\delta) \rightarrow (*e',\mu',\delta')}$$

(REF)

(I) $$(\&(a,1),\mu,\delta) \rightarrow ((a,0),\mu,\delta)$$

(II) $$(\& * (a,0),\mu,\delta) \rightarrow ((a,0),\mu,\delta)$$

(III) $$\frac{(e,\mu,\delta) \rightarrow (e',\mu',\delta')}{(\& * e,\mu,\delta) \rightarrow (\& * e',\mu',\delta')}$$

(OFFSET)

(I) $$\frac{n \text{ an integer}}{(((i,j),0)+n,\mu,\delta) \rightarrow (((i,j+n),0),\mu,\delta)}$$

(II) $$\frac{(e,\mu,\delta) \rightarrow (e',\mu',\delta')}{(((i,j),0)+e,\mu,\delta) \rightarrow (((i,j),0)+e',\mu',\delta')}$$

(III) $$\frac{(e_1,\mu,\delta) \rightarrow (e',\mu',\delta')}{(e_1+e_2,\mu,\delta) \rightarrow (e'+e_2,\mu',\delta')}$$

(APPLY)

(I) $$((\lambda x_1,\ldots,x_n.\, e)(v_1,\ldots,v_n),\mu,\delta) \rightarrow ([v_1,\ldots,v_n/x_1,\ldots,x_n]e,\mu,\delta)$$

(II) $$\frac{(e,\mu,\delta) \rightarrow (e',\mu',\delta')}{(e(e_1,\ldots,e_n),\mu,\delta) \rightarrow (e'(e_1,\ldots,e_n),\mu',\delta')}$$

(III) $$\frac{(e_i,\mu,\delta) \rightarrow (e_i',\mu',\delta') \quad 1 \leq i \leq n}{\begin{array}{l}((\lambda x_1,\ldots,x_n.\, e)(v_1,\ldots,v_{i-1},e_i,\ldots,e_n),\mu,\delta) \rightarrow \\ \qquad ((\lambda x_1,\ldots,x_n.\, e)(v_1,\ldots,v_{i-1},e_i',\ldots,e_n),\mu',\delta')\end{array}}$$

(UPDATE)

(I) $$\frac{a \in dom(\mu) \text{ and } \mu(a) \neq \textbf{dead}}{((a,1) := v,\mu,\delta) \rightarrow (v,\mu[a := v],\delta)}$$

(II) $$\frac{(e,\mu,\delta) \rightarrow (e',\mu',\delta')}{((a,1) := e,\mu,\delta) \rightarrow ((a,1) := e',\mu',\delta')}$$

(III) $$\frac{a \in \mathit{dom}(\mu) \text{ and } \mu(a) \neq \mathbf{dead}}{(*(a,0) := v,\mu,\delta) \rightarrow (v,\mu[a := v],\delta)}$$

(IV) $$\frac{(e,\mu,\delta) \rightarrow (e',\mu',\delta')}{(*(a,0) := e,\mu,\delta) \rightarrow (*(a,0) := e',\mu',\delta')}$$

(V) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(*e_1 := e_2,\mu,\delta) \rightarrow (*e_1' := e_2,\mu',\delta')}$$

(BIND)

(I) $$(\mathbf{let}\ x = v\ \mathbf{in}\ e,\mu,\delta) \rightarrow ([v/x]e,\mu,\delta)$$

(II) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2,\mu,\delta) \rightarrow (\mathbf{let}\ x = e_1'\ \mathbf{in}\ e_2,\mu',\delta')}$$

(BINDVAR)

(I) $$\frac{(i,0) \notin \mathit{dom}(\mu)}{\begin{array}{l}(\mathbf{letvar}\ x := v\ \mathbf{in}\ e,\mu,0) \rightarrow \\ \qquad\qquad (\mathbf{letvar}\ x := v\ \mathbf{in}\ [((i,0),1)/x]e,\mu[(i,0) := v],1)\end{array}}$$

(II) $$\frac{(i,0) \in \mathit{dom}(\mu) \text{ and } (i,0) \text{ the last non-dead cell}}{(\mathbf{letvar}\ x := v_1\ \mathbf{in}\ v_2,\mu,1) \rightarrow (v_2,\mu[(i,0) := \mathbf{dead}],0)}$$

(III) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(\mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2,\mu,\delta) \rightarrow (\mathbf{letvar}\ x := e_1'\ \mathbf{in}\ e_2,\mu',\delta')}$$

(IV) $$\frac{(e,\mu,\delta-1) \rightarrow (e',\mu',\delta')\ \ (\delta > 0)}{(\mathbf{letvar}\ x := v\ \mathbf{in}\ e,\mu,\delta) \rightarrow (\mathbf{letvar}\ x := v\ \mathbf{in}\ e',\mu',\delta'+1)}$$

(BINDARR)

(I) $$\frac{n \text{ a positive integer and } (i,0) \notin \mathit{dom}(\mu)}{\begin{array}{l}(\mathbf{letarr}\ x[n]\ \mathbf{in}\ e,\mu,0) \rightarrow (\mathbf{letarr}\ x[n]\ \mathbf{in}\ [((i,0),0)/x]e, \\ \qquad\qquad \mu[(i,0),\ldots,(i,n-1) := \mathbf{uninit},\ldots,\mathbf{uninit}],1)\end{array}}$$

(II) $$\frac{(i,n-1) \in \mathit{dom}(\mu) \text{ and } (i,n-1) \text{ the last non-dead cell}}{\begin{array}{l}(\mathbf{letarr}\ x[n]\ \mathbf{in}\ v,\mu,1) \rightarrow \\ \qquad\qquad (v,\mu[(i,0),\ldots,(i,n-1) := \mathbf{dead},\ldots,\mathbf{dead}],0)\end{array}}$$

(III) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(\textbf{letarr } x[e_1] \textbf{ in } e_2,\mu,\delta) \rightarrow (\textbf{letarr } x[e_1'] \textbf{ in } e_2,\mu',\delta')}$$

(IV) $$\frac{(e,\mu,\delta-1) \rightarrow (e',\mu',\delta') \ \ (\delta > 0)}{(\textbf{letarr } x[n] \textbf{ in } e,\mu,\delta) \rightarrow (\textbf{letarr } x[n] \textbf{ in } e',\mu',\delta'+1)}$$

(LOOP)

(I) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{\begin{array}{l}(\textbf{while } e_1 \textbf{ do } e_2,\mu,\delta) \rightarrow \\ \qquad\qquad (\textbf{if } e_1' \textbf{ then } e_2; \textbf{while } e_1 \textbf{ do } e_2 \textbf{ else unit},\mu',\delta')\end{array}}$$

(BRANCH)

(I) $$\frac{n \text{ a nonzero integer}}{(\textbf{if } n \textbf{ then } e_1 \textbf{ else } e_2,\mu,\delta) \rightarrow (e_1,\mu,\delta)}$$

(II) $$(\textbf{if } 0 \textbf{ then } e_1 \textbf{ else } e_2,\mu,\delta) \rightarrow (e_2,\mu,\delta)$$

(III) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3,\mu,\delta) \rightarrow (\textbf{if } e_1' \textbf{ then } e_2 \textbf{ else } e_3,\mu',\delta')}$$

(COMPOSE)

(I) $$(v; e,\mu,\delta) \rightarrow (e,\mu,\delta)$$

(II) $$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(e_1; e_2,\mu,\delta) \rightarrow (e_1'; e_2,\mu',\delta')}$$

Meta variable v and x range over *values* and *identifiers*, respectively. The understanding in rules like DEREF, REF, etc. is that if there are transitions on e and v or at least one specific syntactic value then e is understood to be all expressions except all values. For instance, DEREF has two rules; (I) defines a transition for pointer type values and (II) defines a transition for all other expressions except values.

Since the lifetime of a memory cell is bounded by the scope in which it is activated, the rules have to keep track of the lifespan of each memory cell. In SOS, this is easy to do, whereas the solution in TS may seem unintuitive. We introduce δ to keep track of the scope information. Notice that in BINDVAR (I), after a cell is allocated for a variable we still keep the *letvar*[1] construct until the body evaluates to a value. When the cell is allocated δ is incremented so that we can understand that this *letvar* instance has actually allocated a cell and now it is evaluating its body. Rules BINDVAR (I) and BINDVAR (IV) show this difference. In BINDVAR (I), the *letvar* expression of the initial configuration has a value v as its e_1 and δ is 0 which means a cell has not been allocated yet. Then a new cell for x is allocated and initialized to v, and δ is incremented by one. In BINDVAR (IV), the initial configuration is the same as the initial configuration of BINDVAR (I) except that the second premise forces δ be greater than 0 which means that this rule is used only to evaluate the body of *letvar*. Keeping the *letvar* construct around after we allocate a cell makes the proof search part of a *letvar* transition unnecessarily long, but introducing a new construct would force us to augment the type system superficially with a new typing rule for this new construct. The evaluation of a program starts with $\delta = 0$ and ends again with $\delta = 0$.

At first glance, one might be tempted to use a variation of *ρ-expressions* to keep track of the cells being activated [WrF91]. This would not be enough by itself, since in PolyC the lifetime of a cell is bounded whereas in [WrF91] a cell has unbounded lifetime.

We assume that memory cells are allocated sequentially from a sufficiently big sequence of cells, where the cells are associated with index numbers in an increasing order. As defined earlier, an address is a pair of segment and offset numbers and it indicates a cell in the memory. When a variable v is created, the cell with the least index number from the non-used part of the sequence is initialized to the value of this variable, and an address $(i, 0)$ corresponding to this cell is added to the domain of μ. Similiarly, when an array x of size n is created then the first n cells from the non-used part of the sequence are initialized to **uninit** and the corresponding addresses $(i, 0), (i, 1), \ldots, (i, n-1)$ are added to the domain of μ. When the scope of

[1] We will focus on *letvar* without mentioning *letarr* separetely; in most cases the same discussion is also valid for *letarr*.

the variable v or the scope of the array x ends then these cells are marked as **dead**, but they are still kept in the domain of μ.

In SOS, a variable declaration and termination are done within a single evaluation rule so that it is easy to know which address is to be marked as **dead**. But in TS, declaration of a variable and termination of it are done via different rules, and the address information is not carried to the next transition. Given the memory model, it is easy to find out the memory cell to be marked when necessary. Simply search through the sequence of cells starting from the high-index numbered end of the used part of the sequence, and the first cell that is not marked as **dead** will correspond to the address of the variable whose scope is ending. We call this cell *the last non-dead cell.* In case of an array of size n, the consecutive n cells starting from *the last non-dead cell* are the ones that will be marked as **dead**. The reason that we have to search for the last non-dead cell is because dead locations are not taken away from the domain of μ. If an expression e creates a variable x for which the cell indexed i is allocated, and if a subexpression of e then creates another variable y, then the cell allocated for y has a higher index j and so j will be marked as **dead** before i since the scope of y ends before the scope of x.

3. Two Examples of Program Evaluation

Figure 3 shows the evaluation derivation of the program

$$\textbf{letvar } x := 1 \textbf{ in letvar } y := x \textbf{ in } y\,.$$

The evaluation in Figure 3 is completed in six transitions. A transition rule name is given inside brackets to indicate the rule used in making the single transition that follows it. For example, the first transition is done using the BINDVAR (I) rule. The second, third, fourth and fifth transitions are done using an instance of BINDVAR (IV). In the proof search, the second transition uses an instance of BINDVAR (III) and CONTENTS, the third transition uses an instance of BINDVAR (I), the fourth transition uses an instance of BINDVAR (IV), and the fifth transition uses

[BINDVAR (I)]
$(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := x\ \textbf{in}\ y, [], 0) \rightarrow$
$\qquad(\textbf{letvar}\ x := 1\ \textbf{in}\ [((i_x, 0), 1)/x]\,\textbf{letvar}\ y := x\ \textbf{in}\ y, [(i_x, 0) := 1], 1)$

[CONTENTS]
$(((i_x, 0), 1), [(i_x, 0) := 1], 0) \rightarrow (1, [(i_x, 0) := 1], 0)$

[BINDVAR (III)]
$(\textbf{letvar}\ y := ((i_x, 0), 1)\ \textbf{in}\ y, [(i_x, 0) := 1], 0) \rightarrow$
$\qquad(\textbf{letvar}\ y := 1\ \textbf{in}\ y, [(i_x, 0) := 1], 0)$

[BINDVAR (IV)]
$(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := ((i_x, 0), 1)\ \textbf{in}\ y, [(i_x, 0) := 1], 1) \rightarrow$
$\qquad(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := 1\ \textbf{in}\ y, [(i_x, 0) := 1], 1)$

[BINDVAR (I)]
$(\textbf{letvar}\ y := 1\ \textbf{in}\ y, [(i_x, 0) := 1], 0) \rightarrow$
$\qquad(\textbf{letvar}\ y := 1\ \textbf{in}\ [(i_y, 0), 1)/y]y, [(i_x, 0) := 1, (i_y, 0) := 1], 1)$

[BINDVAR (IV)]
$(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := 1\ \textbf{in}\ y, [(i_x, 0) := 1], 1) \rightarrow$
$\qquad(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := 1\ \textbf{in}\ ((i_y, 0), 1), [(i_x, 0) := 1, (i_y, 0) := 1], 2)$

[CONTENTS]
$(((i_y, 0), 1), [(i_x, 0) := 1, (i_y, 0) := 1], 0) \rightarrow (1, [(i_x, 0) := 1, (i_y, 0) := 1], 0)$

[BINDVAR (IV)]
$(\textbf{letvar}\ y := 1\ \textbf{in}\ ((i_y, 0), 1), [(i_x, 0) := 1, (i_y, 0) := 1], 1) \rightarrow$
$\qquad(\textbf{letvar}\ y := 1\ \textbf{in}\ 1, [(i_x, 0) := 1, (i_y, 0) := 1], 1)$

[BINDVAR (IV)]
$(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := 1\ \textbf{in}\ ((i_y, 0), 1), [(i_x, 0) := 1, (i_y, 0) := 1], 2) \rightarrow$
$\qquad(\textbf{letvar}\ x := 1\ \textbf{in letvar}\ y := 1\ \textbf{in}\ 1, [(i_x, 0) := 1, (i_y, 0) := 1], 2)$

Figure 3. Sample Program Derivation, continued next page

[BINDVAR (II)]

$$(\textbf{letvar}\ y := 1\ \textbf{in}\ 1, [(i_x, 0) := 1, (i_y, 0) := 1], 1) \to$$
$$(1, [(i_x, 0) := 1, (i_y, 0) := \textbf{dead}], 0)$$

[BINDVAR (IV)]

$$(\textbf{letvar}\ x := 1\ \textbf{in}\ \textbf{letvar}\ y := 1\ \textbf{in}\ 1, [(i_x, 0) := 1, (i_y, 0) := 1], 2) \to$$
$$(\textbf{letvar}\ x := 1\ \textbf{in}\ 1, [(i_x, 0) := 1, (i_y, 0) := \textbf{dead}], 1], 1)$$

[BINDVAR (II)]

$$(\textbf{letvar}\ x := 1\ \textbf{in}\ 1, [(i_x, 0) := 1, (i_y, 0) := \textbf{dead}], 1) \to$$
$$(1, [(i_x, 0) := \textbf{dead}, (i_y, 0) := \textbf{dead}], 0)$$

Figure 4. Sample Program Derivation, cont.

an instance of BINDVAR (II). The final transition is done with BINDVAR (II). So the *letvar* expression evaluates to 1.

Now let's turn back to the well-typed program

$$\textbf{letvar}\ x := \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&y\ \textbf{in}(*\,x)(3)$$

of Chapter I Section 2, in which the location of y escaped from its scope via the & operator and we inferred the type *int* for this program. Figure 5 shows how this program gets stuck due to dereferencing a **dead** cell.

The notation $\not\to$ denotes the stuck condition of a rule instance. In the sixth transition, $*((i_y, 0), 0)$ attempts to derefence a **dead** location, which causes the evalution to get stuck because there is no possible transition that can be made. The first three transitions are done with BINDVAR (III), where in the proof search the first transition uses an instance of BINDVAR (I), the second transition uses the instances of BINDVAR (IV) and REF, and the third transition uses an instance of BINDVAR (II). The fourth transition is done with BINDVAR (I), because $((i_y, 0), 0)$ is a pointer, which is a syntactic value and δ is 0. The fifth transition is done with BINDVAR (III), where the instances of APPLY and CONTENTS are used in the proof search.

[BINDVAR (I)]
$(\textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&y, [], 0) \rightarrow$
$(\textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ [((i_y,0),1)/y]\,\&y, [(i_y,0) := \lambda z.z], 1)$

[BINDVAR (III)]
$(\textbf{letvar}\, x\ := \ \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&y\ \textbf{in}\ (*\,x)(3), [], 0) \rightarrow$
$(\textbf{letvar}\, x\ := \ \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&((i_y,0),1)\ \textbf{in}(*\,x)(3), [(i_y,0) := \lambda z.z], 1)$

[REF]
$(\&((i_y,0),1), [(i_y,0) := \lambda z.z], 0) \rightarrow (((i_y,0),0), [(i_y,0) := \lambda z.z], 0)$

[BINDVAR (IV)]
$(\textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&((i_y,0),1), [(i_y,0) := \lambda z.z], 1) \rightarrow$
$(\textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ ((i_y,0),0), [(i_y,0) := \lambda z.z], 1)$

[BINDVAR (III)]
$(\textbf{letvar}\, x\ := \ \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ \&((i_y,0),1)\ \textbf{in}(*\,x)(3), [(i_y,0) := \lambda z.z], 1) \rightarrow$
$(\textbf{letvar}\, x\ := \ \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ ((i_y,0),0)\ \textbf{in}(*\,x)(3), [(i_y,0) := \lambda z.z], 1)$

[BINDVAR (II)]
$(\textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ ((i_y,0),0), [(i_y,0) := \lambda z.z], 1) \rightarrow$
$(((i_y,0),0), [(i_y,0) := \textbf{dead}], 0)$

[BINDVAR (III)]
$(\textbf{letvar}\, x\ := \ \textbf{letvar}\ y := \lambda z.z\ \textbf{in}\ ((i_y,0),0)\ \textbf{in}(*\,x)(3), [(i_y,0) := \lambda z.z], 1) \rightarrow$
$\textbf{letvar}\ x := ((i_y,0),0)\ \textbf{in}\ (*\,x)(3), [(i_y,0) := \textbf{dead}], 0)$

[BINDVAR (I)]
$\textbf{letvar}\ x := ((i_y,0),0)\ \textbf{in}\ (*\,x)(3), [(i_y,0) := \textbf{dead}], 0) \rightarrow$
$\textbf{letvar}\ x := ((i_y,0),0)\ \textbf{in}\ [((i_x,0),1)/x]\,(*\,x)(3), [(i_y,0) := \textbf{dead}, (i_x,0) := ((i_y,0),0)], 1)$

Figure 5. Sample Stuck Program, continued next page

[CONTENTS]
$(((i_x, 0), 1), [(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 0) \rightarrow$
$(((i_y, 0), 0), [(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 0)$

[APPLY]
$((* y)(3), [(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 0) \rightarrow$
$((* ((i_y, 0), 0))(3), [(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 0)$

[BINDVAR (III)]
$\textbf{letvar } x := ((i_y, 0), 0) \textbf{ in } (* ((i_x, 0), 1))(3), [(i_y, 0) := \textbf{dead},$
$(i_x, 0) := ((i_y, 0), 0)], 1) \rightarrow \textbf{letvar } x := ((i_y, 0), 0) \textbf{ in } (* ((i_y, 0), 0))(3),$
$[(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 1)$

$(* ((i_y, 0), 0)), [(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 0) \not\rightarrow$

$((* ((i_y, 0), 0))(3), [(i_y, 0) := \textbf{dead}, (i_x, 0) := ((i_y, 0), 0)], 0) \not\rightarrow$

$\textbf{letvar } x := ((i_y, 0), 0) \textbf{ in } [((i_x, 0), 1)/x] (* ((i_y, 0), 0))(3), [(i_y, 0) := \textbf{dead},$
$(i_x, 0) := ((i_y, 0), 0)], 1) \not\rightarrow$

Figure 6. Sample Stuck Program, cont.

4. The LOOP Rule

In the preliminary design of the transition semantics of PolyC, we developed three rules, given below, to specify the transitions for the *while-do* construct.

(LOOP)

(I) $$\frac{(e_1, \mu, \delta) \rightarrow (n, \mu', \delta') \text{ (n a nonzero integer)}}{(\textbf{while } e_1 \textbf{ do } e_2, \mu, \delta) \rightarrow (e_2; \textbf{while } e_1 \textbf{ do } e_2, \mu', \delta')}$$

(II) $$\frac{(e_1, \mu, \delta) \rightarrow (0, \mu', \delta')}{(\textbf{while } e_1 \textbf{ do } e_2, \mu, \delta) \rightarrow (\textbf{unit}, \mu', \delta')}$$

$$\text{(III)} \qquad \frac{(e_1, \mu, \delta) \rightarrow (e_1', \mu', \delta')}{(\textbf{while } e_1 \textbf{ do } e_2, \mu, \delta) \rightarrow (\textbf{while } e_1' \textbf{ do } e_2, \mu', \delta')}$$

Gunter develops a transition semantics for an imperative programming language called Simple Imperative Programming Language (SIPL), and rules (III) and (II) above are closely similiar to Gunter's [Gun92] . There is a subtle difference though: e_1 of **while** e_1 **do** e_2 is not evaluated explicitly in Gunter's system but its value is found by a meaning function in one step. In our system we explicitly evaluate e_1 and for this reason a third rule had to be added to the system as shown above. But in a short time we realized that this third rule was faulty. Assume in an evaluation of a program we reach the point of evaluating the expression,

$$\textbf{while } (a, 1) := ((a, 1) + 1); 1 \textbf{ do } e\,,$$

which increments the value stored in address a and then evaluates the body e. This is an infinite loop, since the value of a sequential composition $e_1; e_2$ is the value of e_2 and, in this program, e_2 is 1 so the condition is always true. In each iteration, $(a, 1) := ((a, 1) + 1); 1$ and e must be evaluated. But this is not achievable with the above rules. The evaluation starts with repeated applications of rule (III) until $(a, 1) := ((a, 1) + 1); 1$ evaluates to the value 1. At this point, the configuration is $(\textbf{while } 1 \textbf{ do } e, \mu, \delta)$ and rule (I) is applied by resulting in the new configuration $(e; \textbf{while } 1 \textbf{ do } e, \mu', \delta')$. After some applications of COMPOSE, e evaluates to a value and then the configuration $(\textbf{while } 1 \textbf{ do } e, \mu'', \delta'')$ is found. This completes the first iteration of the loop; but notice that we have lost the original program: **while** 1 **do** e is different than **while** $(a, 1) := ((a, 1) + 1); 1$ **do** e.

To fix this error, we developed the rule below by using a continuation instead of the three rules:

$$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{\begin{array}{l}(\textbf{while } e_1 \textbf{ do } e_2,\mu,\delta) \rightarrow \\ \qquad ((\lambda x.\textbf{if } x \textbf{ then} \\ \qquad\qquad\quad e_2; \textbf{while } e_1 \textbf{ do } e_2 \\ \qquad\qquad \textbf{else} \\ \qquad\qquad\quad \textbf{unit} \\ \qquad\ )\, e_1',\mu',\delta')\end{array}}$$

In this rule, the λ abstraction is a continuation. We simplify the rule by $\beta -$ *reducing* the application of the continuation to e_1' and arrive at the rule below.

$$\frac{(e_1,\mu,\delta) \rightarrow (e_1',\mu',\delta')}{(\textbf{while } e_1 \textbf{ do } e_2,\mu,\delta) \rightarrow (\textbf{if } e_1' \textbf{ then } e_2; \textbf{while } e_1 \textbf{ do } e_2 \textbf{ else unit},\mu',\delta')}$$

This is the rule for loop construct in the present system.

C. CONCLUSION

Although we have a better handle on the progress of the evaluations of programs, we face an increase in the number of transition rules in the system. When we want to add the binary operations to the language, the number of rules increases greatly. One possible effect of this is that proofs might be complicated and unnecessarily long.

V. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

1. Type Inference Algorithm

We have presented an ML-style type inference algorithm called W_c based on Milner's algorithm W [Mil78] [DaM82]. An implementation of W_c has been given in Appendix as part of an interpreter of PolyC. We expect a correctness proof of W_c be straightforward but it is beyond the scope of this thesis.

2. The Transition Semantics

An imperative programming language with first class pointers should have a stronger property of type soundness than the subject reduction property; i.e., if a closed term has type τ, then the evaluation of that term yields a value of type τ if evaluation terminates successfully. For this reason, Smith and Volpano prove soundness of the PolyC type system by formulating the evaluation rules of PolyC's natural semantics as an instance of a recursive function called *eval* [SmV96a]. But this proof seems to be slightly informal. To establish a basis for a more formal proof, we developed a transition semantics for PolyC and have presented it in this thesis. We believe that a transition semantics exposes more information about the course of an evaluation, thus making it possible to give more rigorous soundness arguments. But a transition semantics tends to introduce a large number of rules in the system, which makes proofs more cumbersome.

B. FUTURE WORK

1. Formal Soundness Proof

Volpano and Smith are currently working on a new soundness proof with respect to natural semantics using partial evaluation trees. Pfenning is also expected to give a soundness proof[1] using the *Elf* programming language, which is based on the *linear logical framework* concept [Pfe96]. We believe a soundness proof of the PolyC type system is possible using the transition semantics given in this thesis as well.

2. Extending PolyC

Extending PolyC with integer and boolean operations is a trivial task, and they have already been included in the interpreter implementation given in Appendix. Polymorphic records and variants, on the other hand, require modifications to the type system and to the type inference algorithm. Ohori [Ohor95] investigates an ML-style polymorphic record calculus in a functional setting by introducing *kinded quantification*, which places restrictions on possible instantiations of type variables. His work is an appealing foundation for labeled records and variants in the PolyC language.

[1] Based on the personal communication during ESOP'96, Linköping Sweden

APPENDIX. SOURCE PROGRAM FOR THE INTERPRETER

1. REMARKS

Developing a type inference algorithm has led to an implementation of W_c to see how it works in practice. Besides type inference we also implemented the natural semantics of Poly C given in [SmV96a] and, as a result, we have created an interpreter for PolyC. During implementation we tried not to go beyond the PolyC calculus and we accomplished this except for SSL lists used in the representation of formal and actual parameters.

Annotations throughout the source code are kept concise by assuming that the reader will have some knowledge about programming language theory and some experience with functional programming.

2. SSL CODE FOR THE INTERPRETER

```
/*******************************************************************
 * This interpreter is written using Synthesizer Generator        *
 * Release 4.2. The code given below is the complete code that    *
 * we have used to generate the interpreter by using the Makefile *
 * given also below. For space efficiency, we put all the files   *
 * together in this appendix, but each file is clearly            *
 * identifiable by the header provided before the beginning of a . *
 * file. The textual appearance order of files in this appendix is *
 * alphabetical except Makefile which is given last. Following are *
 * the files:                                                     *
 *                                                                *
 *   assign.ssl            infer.ssl           lex.ssl            *
 *   assign_infer.ssl      int.ssl             pair.ssl           *
 *   bool.ssl              int_infer.ssl       pair_infer.ssl     *
 *   bool_infer.ssl        lambda.ssl          real.ssl           *
 *   eval.ssl              lambda_infer.ssl    real_infer.ssl     *
 *   explist.ssl           let.ssl             while.ssl          *
 *   id.ssl                let_infer.ssl       while_infer.ssl    *
 *   if.ssl                letarr.ssl          Makefile           *
 *   if_infer.ssl          letarr_infer.ssl                     *
 *                                                              *
 * Naming of files are intended to be informative what is in there; *
 * for instance bool.ssl gives the required definitions like      *
 * abstract syntax, minimal paranthesization, unparsing rules,    *
 * template commands and concrete input syntax of boolean         *
 * operations. Type inference for these operations (constructs) is *
 * in bool_infer.ssl.                                             *
 *                                                                *
 * It should be noted one more time that this interpeter extends  *
 * Poly C [SmV96] with real type and integer and bool operations. *
 *                                                                *
 *****************************************************************/



/*****************************************************************
 * File Name : assign.ssl                                        *
 * Purpose   : Definitions for Compose, Assign, AddrOf, Deref,   *
 *             Unit, Dead, Uninit, InvalidAddr constructors of   *
 *             exp phylum.                                       *
 *****************************************************************/

/* InvalidAddr is returned as a result of a memory lookup */

/* Abstract syntax ------------------------------------------- */
exp :   Compose (exp exp)
    |   Assign (exp exp)
    |   AddrOf (exp)
    |   Deref (exp)
    |   Unit()
    |   Dead, Uninit, InvalidAddr ()
    ;

/* Minimal parenthesization ---------------------------------- */
exp :   Compose PP2(0)
    |   Assign PP2(0)
    |   AddrOf PP1(0)
    |   Deref  PP1(0)
    ;
```

```
/* Unparsing ----------------------------------------------------- */

/*
 * In [SmV96], * is used for dereferencing. But in this
 * implementation we use ! for dereferencing and * for integer
 * multiplication.
*/
exp :   Compose      [ ^ ::= @ "; %n" @ ]
    |   Assign       [ ^ ::= @ " %S(PUNCTUATION::=%S) " @ ]
    |   AddrOf       [ ^ ::= "%S(OPERATOR:&%S)" @ ]
    |   Deref        [ ^ ::= "%S(OPERATOR:!%S)" @ ]
    |   Unit         [ ^ ::= "%S(KEYWORD:unit%S)" ]
    |   Dead         [ ^ ::= "%S(KEYWORD:dead%S)" ]
    |   Uninit       [ ^ ::= "%S(KEYWORD:uninit%S)" ]
    |   InvalidAddr  [ ^ ::= "%S(KEYWORD:invalid%S) %S
                                   (KEYWORD:address%S)" ]
   ;

/* Template commands --------------------------------------------- */
transform exp
   on ";" <exp>: Compose(<exp>, <exp>),
   on "e;<exp>" e when (e != <exp>): Compose(e, <exp>),
   on "<exp>;e" e when (e != <exp>): Compose(<exp>, e),
   on ":=" <exp> : Assign(<exp>, <exp>),
   on "&" <exp> : AddrOf(<exp>),
   on "!" <exp>: Deref(<exp>),
   on "!"  e when (e != <exp>): Deref(e)
   ;

/* Concrete input syntax ----------------------------------------- */
Exp ::= (Exp ASSIGN Exp) {$$.abs = Assign(Exp$2.abs, Exp$3.abs);}
    |   (Exp ’;’ Exp) {$$.abs = Compose(Exp$2.abs, Exp$3.abs);}
    |   (’!’ Exp) {Exp$1.abs = Deref(Exp$2.abs);}
    |   (’&’ Exp) {$$.abs = AddrOf(Exp$2.abs);}
    |   (UNIT) {Exp.abs = Unit;}
    ;



/*******************************************************************
 * File Name : assign_infer.ssl                                    *
 * Purpose   : Type inference for the cons'tors given in assign.ssl  *
 *******************************************************************/
exp :   Unit    {
          exp.typeAssignment = UnitType;
          exp.S = exp.s;
          exp.partial = false;
        }
    |   Dead    {
          exp.typeAssignment = NullType;
          exp.S = FailSubst;
          exp.partial = false;
        }
    |   Uninit  {
          exp.typeAssignment = UniversalType;
          exp.S = exp.s;
          exp.partial = false;
        }
    |   InvalidAddr {
          exp.typeAssignment = UniversalType;
          exp.S = exp.s;
          exp.partial = false;
        }
    |   Deref     {
          local TYPEVAR beta;
          beta = WeakVar(newsymi());
          exp$2.typeEnv = exp$1.typeEnv;
          exp$2.letvars = exp$1.letvars;
          exp$2.s = exp$1.s;
          exp$1.S = Unify(RefType(TypeVar(beta)),
                    exp$2.typeAssignment, exp$2.S);
          exp$1.typeAssignment= ApplySubstToTypeExp(exp$1.S, TypeVar(
                                beta));
          exp$1.partial = exp$2.partial;
          exp$2.sv = exp$1.sv;
          exp$2.encl = exp$1.encl;
          exp$2.top = exp$1.top;
        }
    |   Assign  {
          exp$2.typeEnv = exp$1.typeEnv;
          exp$2.letvars = exp$1.letvars;
          exp$2.s = exp$1.s;
          exp$3.typeEnv = ApplySubstToTypeEnv(exp$2.S, exp$1.typeEnv);
```

```
     exp$3.letvars = exp$1.letvars;
     exp$3.s =
         with(exp$2) (
           Ident(Identifier(i)) : exp$2.S,
           Deref(e)          : exp$2.S,
           Subscript(*,*)    : exp$2.S,
           VoidExp()            : exp$2.S,
           default : FailSubst
         );

     exp$1.typeAssignment =
            ApplySubstToTypeExp(exp$1.S, exp$2.typeAssignment);
     exp$1.S =
       with(exp$2) (
          Ident(Identifier(i)) :
             InLVList(Identifier(i), exp$1.letvars) ?
             Unify(InstScheme(LookupInTypeEnv(i,
             exp$1.typeEnv)),exp$3.typeAssignment,exp$3.S)
             : FailSubst,   /* not a letvar id */
          VoidExp() :
             Unify(exp$2.typeAssignment,exp$3.typeAssignment,
             exp$3.S),
          Deref(e) :
             Unify(exp$2.typeAssignment,exp$3.typeAssignment,
             exp$3.S),
          Subscript(*,*) :
             Unify(exp$2.typeAssignment,exp$3.typeAssignment,
             exp$3.S),
          default : FailSubst
         );
     exp$1.partial = exp$2.partial || exp$3.partial;
     exp$3.sv = exp$1.sv;
     exp$2.sv = exp$1.sv;
     exp$3.encl = exp$1.encl;
     exp$2.encl = exp$1.encl;
     exp$2.top = false;
     exp$3.top = exp$1.top;
   }
|  AddrOf    {
    local TYPEEXP tau;
    exp$2.typeEnv = exp$1.typeEnv;
    exp$2.letvars = exp$1.letvars;
    exp$2.s = exp$1.s;
    exp$1.S =
       with(exp$2) (
          Ident(Identifier(i)) :
             InLVList(Identifier(i),exp$1.letvars)?
           Unify(TypeVar(WeakVar(newsymi())),tau,exp$1.s)
           : FailSubst, /* not a letvar id */
          VoidExp()       : exp$2.S,
          Deref(*)        : exp$2.S,
          Subscript(*, *)  : exp$2.S,
          default         : FailSubst
        );

    exp$1.typeAssignment = RefType(tau);
    exp$1.partial = exp$2.partial;

    tau =
       with(exp$2)(
          Ident(Identifier(i)) :
         InstScheme(LookupInTypeEnv(i,exp$1.typeEnv)),
          VoidExp() : TypeVar(WeakVar(newsymi())),
          Deref(*)  : exp$2.typeAssignment,
           Subscript(*, *) : exp$2.typeAssignment,
          default : NullType
       );

    exp$2.sv = exp$1.sv;
    exp$2.encl = exp$1.encl;
    exp$2.top = exp$1.top;

    }
| Compose {
    exp$2.typeEnv = exp$1.typeEnv;
    exp$2.letvars = exp$1.letvars;
    exp$2.s = exp$1.s;
    exp$3.s = exp$2.S;
    exp$3.letvars = exp$1.letvars;
    exp$3.typeEnv = ApplySubstToTypeEnv(exp$2.S,
                exp$1.typeEnv);
    exp$1.S = exp$3.S;
    exp$1.typeAssignment = exp$3.typeAssignment;
    exp$1.partial = exp$2.partial || exp$3.partial;
```

```
         exp$3.sv = exp$1.sv;
         exp$2.sv = exp$1.sv;
         exp$3.encl = exp$1.encl;
         exp$2.encl = exp$1.encl;
         exp$2.top = false;
         exp$3.top = exp$1.top;
       }
    ;


exp :  Deref  {in TypeErrors on (exp$1.S == FailSubst &&
               exp$2.S != FailSubst);} [ TypeErrors @ : "Deref%n"^ ]
    |  Assign {in TypeErrors on (exp$1.S == FailSubst &&
               exp$2.S != FailSubst && exp$3.S != FailSubst);}
               [ TypeErrors @ : "Assign%n"^^ ]
    |  AddrOf  {in TypeErrors on (exp$1.S == FailSubst &&
                exp$2.S != FailSubst);} [ TypeErrors @ : "AddrOf%n"^ ]
    ;



/*********************************************************************
* File Name : bool.ssl                                              *
* Purpose   : Boolean operations.                                   *
*********************************************************************/

/* Abstract syntax ------------------------------------------------ */
exp :  Not(exp)
    |  And, Or, Equal, NotEqual(exp exp)
    ;

/* Minimal parenthesization --------------------------------------- */
exp :  Not   PP1(9)
    |  And   PP2(3)
    |  Or   PP2(2)
    |  Equal   PP2(4)
    |  NotEqual PP2(4)
    ;

/* Unparsing ------------------------------------------------------ */
exp :  Not   [^ ::= "%S(PUNCTUATION:" lp "%S(OPERATOR:%<not>%S)" @
                "%S(PUNCTUATION:" rp "%S)"]
    | And  [^ ::= "%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:&&%S) "
             @ "%S(PUNCTUATION:" rp "%S)"]
    | Or   [^ ::= "%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:||%S) "
             @ "%S(PUNCTUATION:" rp "%S)"]
    | Equal [^ ::= "%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:=%S) "
              @ "%S(PUNCTUATION:" rp "%S)"]
    | NotEqual  [^ ::= "%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:
                  %<ne>%S) " @ "%S(PUNCTUATION:" rp "%S)"]
    ;

/* Template commands ---------------------------------------------- */
transform exp
   on "~" <exp> : Not(<exp>),
   on "&&" <exp> : And(<exp>, <exp>),
   on "||" <exp> : Or(<exp>, <exp>),
   on "=" <exp> : Equal(<exp>, <exp>),
   on "<>" <exp> : NotEqual(<exp>, <exp>)
   ;

/* Concrete input syntax ------------------------------------------ */
Exp ::=   ('~' Exp)   { Exp$1.abs = Not(Exp$2.abs); }
    |   (Exp LOGICALAND Exp)
            { Exp$1.abs = And(Exp$2.abs, Exp$3.abs); }
    |   (Exp LOGICALOR Exp)
            { Exp$1.abs = Or(Exp$2.abs, Exp$3.abs); }
    |   (Exp '=' Exp prec '=')
            { Exp$1.abs = Equal(Exp$2.abs, Exp$3.abs); }
    |   (Exp NOTEQUAL Exp prec NOTEQUAL)
            { Exp$1.abs = NotEqual(Exp$2.abs, Exp$3.abs); }
    ;


/*********************************************************************
* File Name : bool_infer.ssl                                        *
* Purpose   : Type inference for the cons'tors given in bool.ssl    *
*********************************************************************/

exp :  Not  {
         exp$2.typeEnv = exp$1.typeEnv;
         exp$2.letvars = exp$1.letvars;
         exp$2.s = exp$1.s;
    exp$1.S = Unify(exp$2.typeAssignment,
          IntType, exp$2.S);
    exp$1.typeAssignment = IntType;
    exp$1.partial = exp$2.partial;
    exp$2.sv = exp$1.sv;
    exp$2.encl = exp$1.encl;
    exp$2.top = exp$1.top;
    }
| And, Or  {
    exp$2.typeEnv = exp$1.typeEnv;
    exp$2.letvars = exp$1.letvars;
    exp$3.letvars = exp$1.letvars;
    exp$2.s = exp$1.s;
    exp$3.s = Unify(exp$2.typeAssignment,
          IntType, exp$2.S);
    exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s,
                exp$1.typeEnv);
    exp$1.S = Unify(exp$3.typeAssignment,
          IntType, exp$3.S);
    exp$1.typeAssignment = IntType;
    exp$1.partial = exp$2.partial || exp$3.partial;
    exp$3.sv = exp$1.sv;
    exp$2.sv = exp$1.sv;
    exp$3.encl = exp$1.encl;
    exp$2.encl = exp$1.encl;
    exp$2.top = false;
    exp$3.top = exp$1.top;
    }
| Equal, NotEqual  {
    exp$2.typeEnv = exp$1.typeEnv;
    exp$2.letvars = exp$1.letvars;
    exp$3.letvars = exp$1.letvars;
    exp$2.s = exp$1.s;
    exp$3.s = exp$2.S;
    exp$3.typeEnv = ApplySubstToTypeEnv(exp$2.S, exp$1.typeEnv);
    exp$1.S = Unify(exp$2.typeAssignment, exp$3.typeAssignment,
              exp$3.S);
    exp$1.typeAssignment = IntType;
    exp$1.partial = exp$2.partial || exp$3.partial;
    exp$3.sv = exp$1.sv;
    exp$2.sv = exp$1.sv;
    exp$3.encl = exp$1.encl;
```

```
            exp$2.encl = exp$1.encl;
            exp$2.top = false;
            exp$3.top = exp$1.top;
          }
    ;

exp :  Not    {in TypeErrors on (exp$1.S == FailSubst &&
                 exp$2.S != FailSubst); } [ TypeErrors @ : "Not%n" ^ ]
     | And, Or  {in TypeErrors on (exp$1.S == FailSubst &&
                   exp$2.S != FailSubst && exp$3.S != FailSubst);}
     | And    [ TypeErrors @ : "And%n" ^ ^ ]
     | Or     [ TypeErrors @ : "Or%n" ^ ^ ]
     | Equal, NotEqual {in TypeErrors on (exp$1.S == FailSubst &&
                         exp$2.S != FailSubst && exp$3.S != FailSubst);}
     | Equal    [ TypeErrors @ : "Equal%n" ^ ^ ]
     | NotEqual[ TypeErrors @ : "NotEqual%n" ^ ^ ]
     ;



/*****************************************************************
 * File Name : eval.ssl                                          *
 * Purpose   : Implements the natural semantics (structured      *
 *             operational semantics) of Poly C wrt the rules    *
 *             given in [SmV96]. User has the option to evaluate *
 *             a program or not by clicking on the button labeled *
 *             eval-on.                                          *
 *             When the evaluation of a program gets stuck due   *
 *             to one of four error cases described in [SmV96]   *
 *             the interpreter returns the partially evaluated   *
 *             program as a result for debugging purposes.       *
 *****************************************************************/

MEMORY  : NullMem()                  [@ : ]
        | MemConcat(LOCATION exp MEMORY) {
                INHSILENCE(exp)
          }  [@ : "%{[" @ " \<rightarrow>" @ "]%o" @ "%}"]
        ;


/* Result of an evaluation                */
```

```
EVAL  : EvalPair(exp MEMORY){
              INHSILENCE(exp)
        }   [^ : "%S(PUNCTUATION:(%S)" @ "%S(PUNCTUATION:,%S) %o"  @
             "%S(PUNCTUATION:)%S)" ]
      ;


/*
 * We have two different array subscript constructors : one returns
 * a value as a result of the evaluation (r-value) and the other
 * returns a Varloc (l-value). Having these two constructors is
 * an efficient way of implementing these two different occurrences.
 * Otherwise, if we had only one constructor that returns Varloc
 * then the result of the evaluation of an expression occuring
 * in r-value context must be checked if the result is a Varloc which
 * must be dereferenced with an extra step.
*/

/* We add basic logical operations to the language. They
 * implement the same C semantics as one would expect.
 * False is denoted by 0 and True is denoted by a non-zero
 * value; a logical operation constructors returns 1 if the
 * result of the operation is True.
*/

EVAL eval (exp e, MEMORY mu) {
   with (e) (
     Varloc(l)  :
          EvalPair(MemoryLookUp(l, mu), mu),
     Sum(e1, e2):
          let EvalPair(v1, mu1) = eval(e1, mu) in (
             let EvalPair(v2, mu2) = eval(e2, mu1) in (
                with (v1) (
                   IntOp(i1): with (v2) (
                      IntOp(i2): EvalPair(IntOp(i1 + i2), mu2),
                      default  : EvalPair(Sum(v1, v2), mu)
                      ),
                   default: EvalPair(Sum(v1,e2), mu)
                ))),
     PtrAdd(e1,e2):
          let EvalPair(v1, mu1) = eval(e1, mu) in (
             let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              Refloc(Loc(s,o)): with (v2) (
                 IntOp(i): EvalPair(Refloc(Loc(s,INTtoSTR(
                           STRtoINT(o) + i))), mu2),
                 default : EvalPair(PtrAdd(v1,v2),mu)
                 ),
              default: EvalPair(PtrAdd(v1, e2), mu)
           ))),
Subscript(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              Refloc(Loc(s,o)):
                     with (v2) (
                        IntOp(i):
                           EvalPair(MemoryLookUp(Loc(s,INTtoSTR(
                           STRtoINT(o)+i)), mu2), mu2),
                        default : EvalPair(Subscript(v1,v2),mu)
                     ),
                        default  : EvalPair(Subscript(v1,e2), mu)
                ))),
SubscriptL(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              Refloc(Loc(s,o)): with (v2) (
                IntOp(i) : EvalPair(Varloc(Loc(s,INTtoSTR(
                           STRtoINT(o)+i))), mu2),
                default  : EvalPair(SubscriptL(v1, v2), mu)
                ),
              default  : EvalPair(SubscriptL(v1, e2), mu)
           ))),

Diff(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                       IntOp(i2): EvalPair(IntOp(i1 - i2), mu2),
                       default: EvalPair(Diff(v1, v2), mu)
                       ),
                default: EvalPair(Diff(v1, e2), mu)
```

```
             ))),
Prod(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                       IntOp(i2): EvalPair(IntOp(i1 * i2), mu2),
                       default: EvalPair(Prod(v1, v2), mu)
                       ),
              default: EvalPair(Prod(v1, e2), mu)
           ))),
LessThan(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                      IntOp(i2): EvalPair(IntOp((i1 < i2) ? 1
                                 : 0), mu2),
                      default: EvalPair(LessThan(v1, v2), mu)
                      ),
              default: EvalPair(LessThan(v1, e2), mu)
            ))),
LessThanOrEqual(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                  IntOp(i2): EvalPair(IntOp((i1 <= i2) ? 1: 0),
                  mu2),
                  default: EvalPair(LessThanOrEqual(v1, v2), mu
                    ),
              default: EvalPair(LessThanOrEqual(v1, e2), mu)
           ))),
GreaterThan(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                   IntOp(i2): EvalPair(IntOp((i1 > i2)? 1 : 0),
                               mu2),
                   default: EvalPair(GreaterThan(v1, v2), mu)
                   ),
              default: EvalPair(GreaterThan(v1, e2), mu)
           ))),
GreaterThanOrEqual(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                  IntOp(i2): EvalPair(IntOp((i1 >= i2) ? 1: 0),
                  mu2),
                  default: EvalPair(GreaterThanOrEqual(v1, v2),
                  mu)
                  ),
              default: EvalPair(GreaterThanOrEqual(v1, e2), mu)
           ))),
Quot(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(i1): with (v2) (
                IntOp(i2): (i2 == 0) ? EvalPair(Quot(v1, v2),mu)
                           : EvalPair(IntOp(i1 / i2), mu2),
                default: EvalPair(Quot(v1, v2), mu)
                ),
              default: EvalPair(Quot(v1, e2), mu)
           ))),
Not(e): let EvalPair(v, mu1) = eval(e, mu) in (
           with (v) (
              IntOp(b): EvalPair(IntOp((b == 0) ? 1: 0), mu1),
              default: EvalPair(Not(v), mu)
            )),
And(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(b1): with (v2) (
                    IntOp(b2): EvalPair(IntOp(((b1 != 0) &&
                                (b2 != 0)) ? 1: 0), mu2),
                    default: EvalPair(And(v1, v2), mu)
                    ),
              default: EvalPair(And(v1, e2), mu)
           ))),
Or(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           with (v1) (
              IntOp(b1): with (v2) (
                   IntOp(b2): EvalPair(IntOp(((b1 != 0) ||
                               (b2 != 0)) ? 1: 0), mu2),
                  default: EvalPair(Or(v1, v2), mu)
                  ),
              default: EvalPair(Or(v1, e2), mu)
           ))),
Equal(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           Value(v1) ? Value(v2) ? EvalPair(IntOp((v1 == v2) ? 1
                                   : 0), mu2)
                                   : EvalPair(Equal(v1, v2), mu)
                      : EvalPair(Equal(v1, e2), mu)
          )),
NotEqual(e1, e2):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = eval(e2, mu1) in (
           Value(v1) ? Value(v2) ? EvalPair(IntOp((v1 != v2) ? 1
                                   : 0), mu2)
                                 : EvalPair(NotEqual(v1, v2), mu)
                      : EvalPair(NotEqual(v1, e2), mu)
          )),
Deref(e1):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        with (v1) (
           Refloc(l): EvalPair(MemoryLookUp(l, mu1), mu1),
           default: EvalPair(Deref(v1), mu)
          )),
Call(e1,a1):
     let EvalPair(v1, mu1) = eval(e1, mu) in (
        let EvalPair(v2, mu2) = EvalList(a1, mu1,
                                   ActualParamListNil()) in (
           with(v1)(
              Lambda(x,e2): with(v2)(
                   Call(Unit, a2): eval(ReplaceWithActuals(
                                    a2, x, e2), mu2),
                   Call(Dead, a2): EvalPair(Call(v1, a2), mu2),
                   default: EvalPair(Call(v1,
                                 v2::ActualParamListNil),mu)
                                             /* never happens */
                      ),
              default: EvalPair(Call(v1, a1), mu)
           ))),
Assign(e1, e2):
   with(e1) (
      Deref(e1):
         let EvalPair(v1, mu1) = eval(e1, mu) in (
            let EvalPair(v2, mu2) = eval(e2, mu1) in (
               with (v1) (
                  Refloc(l):  with (MemoryLookUp(l, mu2)) (
                  Dead: EvalPair(Assign(Deref(Dead), e2),
                                mu),
                  default: Value(v2) ?
                     EvalPair(v2, UpdateMemory(l, v2, mu2))
                           : EvalPair(Assign(Varloc(l), v2), mu)
                         ),
                      default:EvalPair(Assign(Deref(v1), e2), mu)
                   ))),
        Subscript(e3, e4):
           let EvalPair(v1, mu1) =
                              eval(SubscriptL(e3,e4), mu) in (
              let EvalPair(v2, mu2) = eval(e2, mu1) in (
                   with (v1) (
                      Varloc(l): Value(v2) ? EvalPair(v2,
                                  UpdateMemory(l, v2, mu2))
                                 : EvalPair(Assign(v1,v2), mu),
                      default  : EvalPair(Assign(v1, e2), mu)
                   ))),
        Varloc(l):
           let EvalPair(v, mu1) = eval(e2, mu) in (
                 Value(v) ? EvalPair(v, UpdateMemory(l, v, mu1))
                           : EvalPair(Assign(e1,v), mu)),
        default: EvalPair(e, mu)
     ),
AddrOf(e1):
   with(e1) (
      Deref(e2):
         let EvalPair(v1, mu1) = eval(e2, mu) in (
            with (v1) (
               Refloc(l): EvalPair(v1,  mu1),
               default  : EvalPair(AddrOf(Deref(v1)), mu)
            )
         ),
      Subscript(e2, e3):
         let EvalPair(v1, mu1) = eval(SubscriptL(e2, e3),mu) in (
            with (v1) (
               Varloc(l): EvalPair(Refloc(l), mu1),
               default  : EvalPair(AddrOf(v1), mu)
            )
          ),
      Varloc(l): EvalPair(Refloc(l), mu),
      default  : EvalPair(e, mu)
   ),
Compose(e1, e2):
   let EvalPair(v, mu1) = eval(e1, mu) in (
      Value(v) ? eval(e2, mu1): EvalPair(Compose(v, e2), mu)
   ),
While(e1, e2):
   let EvalPair(v1, mu1) = eval(e1, mu) in (
      with (v1) (
         IntOp(n): (n != 0) ?
                    let EvalPair(v2, mu2) = eval(e2, mu1) in (
                        Value(v2) ? eval(e, mu2)
                                 : EvalPair(While(v1, v2), mu))
                            : EvalPair(Unit, mu1),
         default: EvalPair(While(v1, e2), mu)
      )),
Cond(e1, e2, e3):
   let EvalPair(v1, mu1) = eval(e1, mu) in (
      with (v1) (
         IntOp(n): eval((n != 0) ? e2: e3, mu1),
         default: eval(Cond(v1, e2, e3), mu)
      )),
LetVar(x, e1, e2):
   let EvalPair(v1, mu1) = eval(e1, mu) in (
      Value(v1) ?
         let l = (newsymi())[2:] in (
            let EvalPair(v2, mu2) =
               eval(ReplaceIn(Varloc(Loc(l,INTtoSTR(0))), x, e2),
               UpdateMemory(Loc(l, INTtoSTR(0)), v1, mu1)) in (
                 Value(v2) ? EvalPair(v2, UpdateMemory(Loc(l,
                 INTtoSTR(0)), Dead, mu2)): EvalPair(v2, mu)))
```

```
                     : EvalPair(LetVar(x, v1, e2), mu)),
     Let(x, e1, e2):
        let EvalPair(v1, mu1) = eval(e1, mu) in (
           Value(v1) ? eval(ReplaceIn(v1, x, e2), mu1)
                    : EvalPair(Let(x, v1, e2), mu)
        ),
     LetArr(x, e1, e2):
        let EvalPair(v1, mu1) = eval(e1, mu) in (
           with(v1) (
              IntOp(n): (n > 0) ?
                 let EvalPair(v2, mu2) = InitializeArray(n, mu1) in (
                    let EvalPair(v3, mu3) = eval(ReplaceIn(v2, x, e2),
                                              mu2) in (
                       Value(v3) ? EvalPair(v3,MarkDead(n, v2, mu3))
                                : EvalPair(v3, mu2)
                 ))
                       : EvalPair(LetArr(x, v1, e2), mu), /* n <= 0*/
              default: EvalPair(LetArr(x, v1, e2), mu)
           )),
     Pair(e1, e2):
        let EvalPair(v1, mu1) = eval(e1, mu) in (
           Value(v1) ? let EvalPair(v2, mu2) = eval(e2, mu1) in (
                          Value(v2) ? EvalPair(Pair(v1, v2), mu2)
                                   : EvalPair(Pair(v1, v2), mu)
                        )
                    : EvalPair(e, mu)
        ),
     default: EvalPair(e, mu)
     )
};

/* Is the expression e a syntactic value? */
BOOL Value(exp e) {
     with(e) (
        Lambda(*,*): true,
        IntOp(*)   : true,
        RealOp(*)  : true,
        Refloc(*)  : true,
        Ident(*)   : true,
        Unit       : true,
        default    : false
     )
```

```
};

/* Replace all free occurrences of  formal parameters given by
 * f in e with the actual parameters given by a.
*/
exp ReplaceWithActuals(actualParamList a, formalParamList f, exp e) {
     with(a) (
        ActualParamListPair(v1, rest1):
           with(f) (
              FormalParamListPair(x, rest2):
                ReplaceIn(v1, x, ReplaceWithActuals(rest1, rest2, e)),
              default: e
           ),
        default: e
     )
};

/* [v/x]e -- replace all free occurrences of x in e by v      */
exp ReplaceIn (exp v, Id x, exp e) {
     with (x) (
        IdNull(): e,
        Identifier(y): ReplaceAux(v, y, e)
     )
};

exp ReplaceAux (exp v, ID id, exp e) {
   with (e) (
      Ident(Identifier(x)): (id == x) ? v: e,
      AddrOf(e1): AddrOf(ReplaceAux(v, id, e1)),
      Assign(e1,e2): Assign(ReplaceAux (v, id, e1),ReplaceAux (v, id
                     ,e2)),
      Deref(e1): Deref(ReplaceAux(v, id, e1)),
      Compose(e1, e2): Compose(ReplaceAux(v, id, e1), ReplaceAux(v,
                       id,e2)),
      Lambda(f, e1): IsFormalParameter(id, f) ? e
                       : Lambda(f, ReplaceAux(v, id, e1)),
      While(e1, e2): While(ReplaceAux(v, id, e1),ReplaceAux(v, id, e2)
                     ),
      Let(Identifier(x), e1, e2):
           (id == x) ? Let(Identifier(x), ReplaceAux(v, id, e1), e2)
                       : Let(Identifier(x), ReplaceAux(v, id, e1),
                         ReplaceAux(v, id, e2)),
LetVar(Identifier(x), e1, e2):
  (id == x) ? LetVar(Identifier(x), ReplaceAux(v, id, e1), e2)
            : LetVar(Identifier(x), ReplaceAux(v, id, e1),
              ReplaceAux(v, id, e2)),
LetArr(Identifier(x), e1, e2):
    (id == x) ? LetVar(Identifier(x), ReplaceAux(v, id, e1),
                                      e2)
              : LetArr(Identifier(x), ReplaceAux(v, id, e1),
                                        ReplaceAux(v, id, e2)),
PtrAdd(e1, e2): PtrAdd(ReplaceAux(v, id, e1), ReplaceAux(v, id,
                e2)),
Subscript(e1, e2): Subscript(ReplaceAux(v, id, e1), ReplaceAux(
                   v, id, e2)),
SubscriptL(e1, e2): SubscriptL(ReplaceAux(v, id, e1),ReplaceAux(
                    v, id, e2)),
Pair(e1, e2): Pair(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)
              ),
Sum(e1, e2): Sum(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)),
Diff(e1, e2): Diff(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2
              )),
Prod(e1, e2): Prod(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)
              ),
Quot(e1, e2): Quot(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)
              ),
LessThan(e1, e2):
  LessThan(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)),
LessThanOrEqual(e1, e2):
  LessThanOrEqual(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)),
GreaterThan(e1, e2):
  GreaterThan(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)),
GreaterThanOrEqual(e1, e2):
  GreaterThanOrEqual(ReplaceAux(v, id, e1), ReplaceAux(v,id,e2)
  ),
Not(e): Not(ReplaceAux(v, id, e)),
And(e1, e2): And(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)),
Or(e1, e2): Or(ReplaceAux(v, id, e1), ReplaceAux(v, id, e2)),
Equal(e1, e2): Equal(ReplaceAux(v, id, e1), ReplaceAux(v,id,e2)
               ),
NotEqual(e1, e2): NotEqual(ReplaceAux(v, id, e1),
                  ReplaceAux(v, id, e2)),
Cond(e1, e2, e3): Cond( ReplaceAux(v, id, e1), ReplaceAux(v, id,
                  e2), ReplaceAux(v, id, e3)),
      Call(e1, l): Call(ReplaceAux(v, id, e1), ReplaceInList(v, id,l)
                   ),
      default: e
   )
};

/* Does id occur in formal parameter list x ? */
BOOL IsFormalParameter(ID id, formalParamList x) {
     with (x) (
       FormalParamListPair(Identifier(v), rest):
           (id == v) ? true: IsFormalParameter(id, rest),
       default: false
     )
};


/* Replace all free occurrences of id in each element e of l */
actualParamList ReplaceInList(exp v, ID id, actualParamList l) {
     with (l) (
        ActualParamListNil: l,
        ActualParamListPair(e, rest):
               ReplaceAux(v, id, e):: ReplaceInList(v, id, rest),
     )
};


/* We evaluate the actual paramaters l1 in order and put the
 * results into another list l2. We use the constructor Call
 * as a placeholder to return the result since it is the only
 * expression constructor with a actualParamList type of argument.
 * The first argument of Call is used to indicate if the
 * evaluation of l1 is completed successfully. If so, we return
 * Unit as the first argument and l2 as the second argument,
 * otherwise we return Dead as the first argument and a partially
 * evaluated list as the second argument.
*/

EVAL EvalList( actualParamList l1, MEMORY mu, actualParamList l2) {
     with(l1) (
       ActualParamListPair(e, rest):
          let EvalPair(v, mu1) = eval(e, mu) in (
             Value(v) ? EvalList(rest, mu1, v::l2)
```

```
                       : EvalPair(Call(Dead, ReverseList(ReverseList(
                          rest) @ v::l2)), mu1)),
       default: EvalPair(Call(Unit,ReverseList(l2)), mu)
    )
};

actualParamList ReverseList(actualParamList l) {
     with(l) (
        ActualParamListPair(v, rest): ReverseList(rest) @
                         ActualParamListPair(v, ActualParamListNil()),
        default: l
     )
};



/* mu[l:=v] -- update(extend) memory mu with binding l:=v    */

MEMORY UpdateMemory (LOCATION l, exp v, MEMORY mu) {
   with (mu) (
      NullMem(): MemConcat(l, v, mu),
      MemConcat(l2, v2, mu2): (l == l2) ? MemConcat(l, v, mu2)
          : MemConcat(l2, v2, UpdateMemory(l, v, mu2)),
   )
};


exp MemoryLookUp (LOCATION l, MEMORY mu) {
   with (mu) (
      MemConcat(l2, v2, mu2): (l == l2) ? v2: MemoryLookUp(l, mu2),
      default: InvalidAddr /*Dereference of a non-existence address */
   )
};

/* Allocate memory cells for the elements of an array of size n and */
/* initialize them to Uninit.*/
EVAL InitializeArray(INT n, MEMORY mu) {
     let l = (newsymi())[2:] in (
        let mu1 = InitializeArrayAux(n - 1, l, UpdateMemory(Loc(l,
                  INTtoSTR(n-1)), Uninit, mu)) in (
           EvalPair(Refloc(Loc(l,INTtoSTR(0))), mu1)
        ))
```

```
};

MEMORY InitializeArrayAux(INT n, SEGMENT s, MEMORY mu) {
     (n == 0) ? mu: InitializeArrayAux(n - 1, s,
                     UpdateMemory(Loc(s,INTtoSTR(n-1)), Uninit, mu))
};


/* Mark the cells allocated for the elements of the array as Dead */
MEMORY MarkDead(INT n, exp e,  MEMORY mu) {
     with(e) (
        Refloc(Loc(s,*)): MarkDeadAux(n, s, mu),
        default  : mu /* should never be reached */
     )
};

MEMORY MarkDeadAux(INT n, SEGMENT s,  MEMORY mu) {
     (n == 0) ? mu: MarkDeadAux(n-1, s,
                     UpdateMemory(Loc(s,INTtoSTR(n-1)), Dead, mu))
};




/*********************************************************************
* File Name : explist.ssl                                           *
* Purpose   : A program is an explist composed of terms.            *
*********************************************************************/

root expList;

/* Abstract syntax ----------------------------------------------- */
term :  Static(exp)
     |  Dynamic(exp)
     ;
list expList;
expList :  ExpListPair(term expList)
        |  ExpListNil()
        ;

/* Minimal parenthesization -------------------------------------- */
term :  Static, Dynamic { exp.precedence = 0; }
```

```
     ;

/* Unparsing ------------------------------------------------------ */
expList : ExpListPair  [ @ : ^ ["%S(PUNCTUATION:;%S)%n%n"] @ ]
        ;

/* Concrete input syntax ------------------------------------------ */
ExpList { synthesized expList abs; };
expList ~ ExpList.abs;
ExpList ::=  (Exp) { ExpList.abs  = Static(Exp.abs) :: ExpListNil(); }
        |   (Exp ';' ExpList) {ExpList$1.abs =
                               Static(Exp.abs) :: ExpList$2.abs; }
        ;




/********************************************************************
* File Name : id.ssl                                                *
* Purpose   : Defines identifiers of the language                   *
********************************************************************/

/* Abstract syntax and unparsing --------------------------------- */
Id :  IdNull()   [ ^ ::= "%S(PLACEHOLDER:<identifier>%S)" ]
   |  Identifier(ID)   [ ^ ::= ^ ]
   ;

/* Concrete input syntax ----------------------------------------- */
id { synthesized Id abs; };
Id ~ id.abs;
id   ::= (ID)   { id.abs = Identifier(ID); }
     |   (IDENTIFIER_PLACEHOLDER)
         { id.abs = IdNull; }
     ;

/* Attribution --------------------------------------------------- */
Id {synthesized ID name;
    synthesized BOOL partial;
   };
Id :  IdNull      {
```

```
         Id.name = "_undeclared";
         Id.partial = true;
      }
    |  Identifier   {
         Id.name = ID;
         Id.partial = false;
       }
    ;


/********************************************************************
* File Name : if.ssl                                                *
* Purpose   : Defines the if-then-else construct                    *
********************************************************************/

/* Abstract syntax ----------------------------------------------- */
exp :  Cond(exp exp exp);

/* Minimal parenthesization -------------------------------------- */
exp :  Cond {
         exp$2.precedence = 0;
         exp$3.precedence = 0;
         exp$4.precedence = 0;
       }
    ;

/* Unparsing ----------------------------------------------------- */
exp :  Cond
         [^ ::= "%t%{%S(KEYWORD:if%S) " @ " %c%S(KEYWORD:then%S) " @
          " %c%S(KEYWORD:else%S) " @ " %b%c%S(KEYWORD:fi%S)%}"]
    ;

/* Template commands --------------------------------------------- */
transform exp
   on "if" <exp>: Cond(<exp>, <exp>, <exp>),
   on "if" e when (e != <exp>): Cond(<exp>, e, <exp>)
   ;

/* Concrete input syntax ----------------------------------------- */
Exp ::=   (IF Exp THEN Exp ELSE Exp FI)
         { Exp$1.abs = Cond(Exp$2.abs, Exp$3.abs, Exp$4.abs); }
    ;


/*******************************************************************
* File Name : if_infer.ssl                                         *
* Purpose   : Type inference for if-then-else construct            *
*******************************************************************/

exp :  Cond {
          exp$2.typeEnv = exp$1.typeEnv;
          exp$2.letvars = exp$1.letvars;
          exp$3.letvars = exp$1.letvars;
          exp$4.letvars = exp$1.letvars;
          exp$2.s = exp$1.s;
          exp$3.s = Unify(exp$2.typeAssignment, IntType, exp$2.S);
          exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s, exp$1.typeEnv);
          exp$4.s = exp$3.S;
          exp$4.typeEnv = ApplySubstToTypeEnv(exp$3.S, exp$1.typeEnv);
          exp$1.S = Unify(exp$3.typeAssignment, exp$4.typeAssignment,
                    exp$4.S);
          exp$1.typeAssignment = exp$3.typeAssignment;
          exp$1.partial = exp$2.partial || exp$3.partial ||
                          exp$4.partial;
          exp$4.sv = exp$1.sv;
          exp$3.sv = exp$1.sv;
          exp$2.sv = exp$1.sv;
          exp$4.encl = exp$1.encl;
          exp$3.encl = exp$1.encl;
          exp$2.encl = exp$1.encl;
          exp$2.top = false;
          exp$3.top = false;
          exp$4.top = false;
       }
    ;

exp :  Cond  {in TypeErrors on (exp$1.S == FailSubst &&
              exp$2.S != FailSubst && exp$3.S != FailSubst &&
              exp$4.S != FailSubst); } [ TypeErrors @ : "If%n" ^ ^ ^ ]
    ;


/*******************************************************************
```

```
* File Name : infer.ssl                                             *
* Purpose   : Implementation of the type inference for              *
*             Poly C. This implementation is based on               *
*             Dennis Volpano's implementation for core ML with      *
*             letvar and first-class refs.                          *
*********************************************************************/

STR foreign newsymi(); /* generate symbols *1, *2, *3 .... */

/* Poly C has only weak type variables.*/
TYPEVAR : WeakVar (STR)         [@ : @ ]
        ;

/* We need this phylum to type the functions of Poly C */
list TYPEEXPLIST;
TYPEEXPLIST : TypeExpListNil()          [@:]
            | TypeExpListPair(TYPEEXP TYPEEXPLIST)
              [ @ : ^ ["%S(OPERATOR: \<times> %S)%o"] @ ]
            ;

TYPEEXP : NullType()          [@ : "?" ]
        | UniversalType()       [@ : "\<bottom>" ]
        | IntType  ()         [@ : "int" ]
        | RealType  ()          [@ : "real" ]
        | UnitType ()         [@ : "unit" ]
        | TypeVar (TYPEVAR)       [@ : @ ]
        | MapType  (TYPEEXPLIST TYPEEXP)[@ : "(" @ "%S(OPERATOR:
          \<rightarrow> %S)%o" @ ")" ]
        | PairType (TYPEEXP TYPEEXP)    [@ : "(" @ " \<times> " @ ")" ]
        | RefType  (TYPEEXP)       [@ : @ " ptr" ]
        ;

TYPESCHEME
   : TypeExp (TYPEEXP)          [@ : @]
   | TypeVarBinding (TYPEVAR TYPESCHEME)    [@ : "\<forall>" @ "." @]
   ;

TYPEEXP TypeExpOfTypeScheme(TYPESCHEME t) {
    with(t) (
        TypeExp(e): e,
        TypeVarBinding(i, s): TypeExpOfTypeScheme(s),
        )
    };

/* Substitutions : Finite functions mapping type variables to types
*  Empty substitution is denoted by IdSubst
*/

SUBST   : FailSubst()       [@ : "FailSubst"]
        | IdSubst()              [@ : ]
        | SubstConcat(TYPEVAR TYPEEXP SUBST)
          [@ : "%{<" @ ":" @ ">%o" @ "%}" ]
        ;

BOOL InSubst(TYPEVAR tyvar, SUBST s) {
    with(s) (
      FailSubst: false,
      IdSubst: false,
      SubstConcat(j, *, sub): j == tyvar ? true : InSubst(tyvar, sub),
    )
};

TYPEEXP LookupInSubst(TYPEVAR tyvar, SUBST s) {
    with(s) (
      FailSubst: NullType,
      IdSubst: UniversalType,
      SubstConcat(j, t, sub): j == tyvar ? t :
           LookupInSubst(tyvar, sub),
      default : tyvar
    )
};

TYPEEXP Ult(TYPEEXP t, SUBST s) {   /* close substitution s for t */
   with (t) (
      TypeVar(v) : InSubst(v, s) ?
            Ult(LookupInSubst(v, s), s) : t,
      default : t
   )
};

TYPEEXP RecRealAux(TYPEEXP t, SUBST s) {
    with (t) (
       TypeVar(v) : let e = LookupInSubst(v, s) in (
         with(e) (
       NullType: t,
       UniversalType: t,
       default: RecRealAux(e, s)
       )
     ),
   MapType(u, w): MapType(RecRealListAux(u, s), RecRealAux(w, s)),
   PairType(u, w): PairType(RecRealAux(u, s), RecRealAux(w, s)),
   RefType(u): RefType(RecRealAux(u, s)),
   default: t
  )
};

TYPEEXPLIST RecRealListAux(TYPEEXPLIST l, SUBST s) {
  with(l) (
   TypeExpListPair(v, l) : RecRealAux(v, s) :: RecRealListAux(l, s),
   default : l
  )
};

TYPEEXP RecReal(TYPEEXP t, SUBST s) {
  with(s) (
    FailSubst: NullType,
    IdSubst: t,
    default: RecRealAux(t, s),
    )
};

SUBST RemoveFromSubst(SUBST s, TYPEVAR id) {
  with (s) (
    FailSubst: FailSubst,
    IdSubst: IdSubst,
    SubstConcat(i, t, sub):
      i == id ? sub : SubstConcat(i, t, RemoveFromSubst(sub, id)),
    )
};

TYPEEXP ApplySubstToTypeVar(SUBST s, TYPEVAR v) {
  with (s) (
    FailSubst: NullType,
    default: let t = LookupInSubst(v, s) in (
     with (t) (
       UniversalType: TypeVar(v),
             default: ApplySubstToTypeExp(s, t)
             )
        ))
};

TYPEEXP ApplySubstToTypeExp(SUBST s, TYPEEXP t) {
    with(s) (
      FailSubst: NullType,
      IdSubst: t,
      default:
        with(t) (
           TypeVar(u): ApplySubstToTypeVar(s, u),
           MapType(t1, t2): MapType(ApplySubstToTypeExpList(s, t1),
                                   ApplySubstToTypeExp(s, t2)),
           PairType(t1, t2): PairType(ApplySubstToTypeExp(s, t1),
                                   ApplySubstToTypeExp(s, t2)),
           RefType(t): RefType(ApplySubstToTypeExp(s, t)),
           default: t
           )
      )
};


TYPEEXPLIST  ApplySubstToTypeExpList(SUBST s, TYPEEXPLIST t) {
    with(t) (
      TypeExpListPair(v, l): ApplySubstToTypeExp(s, v)::
                             ApplySubstToTypeExpList(s, l),
      default : t
      )
};


TYPESCHEME ApplySubstToTypeScheme(SUBST s, TYPESCHEME t) {
    with(t) (
        TypeExp(e): TypeExp(ApplySubstToTypeExp(s, e)),
        TypeVarBinding(i, u):
           TypeVarBinding(i, ApplySubstToTypeScheme(RemoveFromSubst(
           s, i), u)),
        )
};
```

```
/*************************************************
*  let/letarr/letvar-bound identifier list        *
**************************************************/

list LVLIST;
LVLIST : LVNil()               [@ : ]
       | LVCons(Id LVLIST)     [@ : @ [", "] @ ]
       ;

LVLIST RemoveFromLVList (Id id, LVLIST l) {
   with (l) (
      LVNil : l,
      LVCons(v as IdNull(), rest) :
         v :: RemoveFromLVList(id, rest),
      LVCons(v, rest) : (v == id) ? rest :
         v :: RemoveFromLVList(id, rest)
   )
};

BOOL InLVList (Id id, LVLIST l) {
   with(l) (
      LVNil : false,
      LVCons(v, rest) : (v == id) ? true : InLVList(id, rest),
      )
};


/*****************************************************
*  variable/identifier list                          *
*****************************************************/
list VLIST;
VLIST :  BVNil()                 [@ : ]
       | BVCons(Id VLIST)       [@ : @ [", "] @ ]
       ;

BOOL InVList (Id id, VLIST l) {
   with(l) (
      BVNil : false,
      BVCons(v, rest) : (v == id) ? true : InVList(id, rest),
      )
};
```

```
/*************************************************
*  static (top level) variable/identifier list   *
*  These identifiers are the ones whose          *
*  declaration satisfies the conditions to become*
*  top level as explained in Chapter I.          *
*************************************************/
list SVLIST;
SVLIST : SVNil()                [@ : ]
       | SVCons(Id SVLIST)      [@ : @ [", "] @ ]
       ;


/*********************
 * Type environments *
 *********************/

TYPEENV   : NullTypeEnv()            [@ : ]
          | TypeEnvConcat(ID TYPESCHEME TYPEENV)
                          [@ : "%{[" @ ":" @ "]%o" @ "%}" ]
   ;

/*
 * RemoveFromTypeEnv
 *
 * Remove entry for id from s.
 * Note: we assume s contains only one entry for id.
 */
TYPEENV RemoveFromTypeEnv(ID id, TYPEENV s) {
    with(s) (
       NullTypeEnv: s,
       TypeEnvConcat(i, t, tail):
          id == i ? tail
                  : TypeEnvConcat(i, t, RemoveFromTypeEnv(id, tail))
       )
};

TYPESCHEME LookupInTypeEnv(ID id, TYPEENV s) {
    with(s) (
       NullTypeEnv: TypeExp(UniversalType),
       TypeEnvConcat(i, t, tail): id == i ? t : LookupInTypeEnv(id,
                                                 tail),
       )
};

TYPEENV ApplySubstToTypeEnv(SUBST s, TYPEENV e) {
    with(s) (
       IdSubst: e,
       FailSubst: e,
       default:
           with(e) (
              NullTypeEnv: NullTypeEnv,
              TypeEnvConcat(i, t, tail):
                  TypeEnvConcat(i, ApplySubstToTypeScheme(s, t),
                  ApplySubstToTypeEnv(s, tail)),
           )
       )
};


/********************************
*  Generate a generic instance *
*********************************/

/* list of type variables */
list TVLIST;
TVLIST :  TVNil()                     [@ : ]
       |  TVCons(TYPEVAR TVLIST)      [@ : @ [", "] @ ]
       ;

/* return all type vars in type exp t */
TVLIST TvarsIn (TYPEEXP t, TVLIST l) {
   with (t) (
      TypeVar(v)      : v :: l,
      MapType(t1,t2)  : TvarsIn(t2,TvarsInList(t1,l)),
      PairType(t1,t2) : TvarsIn(t2,TvarsIn(t1,l)),
      RefType(t)      : TvarsIn(t, l),
      default         : l
      )
};


TVLIST TvarsInList (TYPEEXPLIST t, TVLIST l) {
   with(t) (
      TypeExpListPair(v, rest) : TvarsInList(rest, TvarsIn(v, l)),
      default : l
   )
};


/* is type var x in type var list? */
BOOL InTVList (TYPEVAR x, TVLIST l) {
   with(l) (
      TVNil : false,
      TVCons(v, rest) : (v == x) ? true : InTVList(x, rest)
   )
};


/* all x members not in y */
TVLIST Bar (TVLIST x, TVLIST y) {
   with (x) (
      TVNil : TVNil,
      TVCons(v,rest) : InTVList(v, y) ? Bar(rest,y)
         : TVCons(v, Bar(rest, y))
   )
};


/* free type vars in scheme */
TVLIST FreeScheme (TYPESCHEME s, TVLIST scvs) {
   with (s) (
      TypeExp(t) : Bar( TvarsIn(t,TVNil), scvs),
      TypeVarBinding(v, rest) : FreeScheme(rest, TVCons(v,scvs)),
   )
};


/* free type vars in type environment */
TVLIST FreeTe (TYPEENV te) {
   with(te) (
      NullTypeEnv: TVNil,
      TypeEnvConcat(i,t,tail): FreeScheme(t,TVNil) @ FreeTe(tail),
   )
};
```

```
/* return a list of noduplicates */
TVLIST Nodups (TVLIST l, TVLIST acc) {
   with(l) (
      TVNil: acc,
      TVCons(v, tail) :
          InTVList(v,acc) ? Nodups(tail, acc)
                          : Nodups(tail, TVCons(v,acc)),
   )
};


TYPESCHEME MkScheme(TVLIST vs, TYPEEXP t) {
   with(vs) (
      TVNil: TypeExp(t),
      TVCons(v, tail):
         TypeVarBinding(v, MkScheme(tail,t)),
   )
};


/* normal closure */
TYPESCHEME Close(TYPEENV a, TYPEEXP t) {
   MkScheme(Bar(Nodups(TvarsIn(t,TVNil),TVNil), FreeTe(a)), t)
};


/* instantiate a scheme */
TYPEEXP InstSchemeAux (TYPESCHEME ts, SUBST s) {
   with(ts) (
      TypeExp(t) : ApplySubstToTypeExp(s, t),
      TypeVarBinding(v, rest) :
         InstSchemeAux(rest, SubstConcat(v,
            TypeVar(WeakVar(newsymi())), s))
   )
};


TYPEEXP InstScheme(TYPESCHEME s) {
   InstSchemeAux(s, IdSubst)
```

```
};



/**************************************
 *  Unification of type expressions  *
 **************************************/

SUBST Unify(TYPEEXP t, TYPEEXP u, SUBST s) {
    s == FailSubst ? FailSubst : Equate(Ult(t, s), Ult(u, s), s)
    };


/* unifies lefthand side of a function space operator */
SUBST UnifyList(TYPEEXPLIST t, TYPEEXPLIST u, SUBST s) {
   (s == FailSubst) ? FailSubst :
   with (t) (
      TypeExpListPair(v1, rest1) :
         with(u) (
       TypeExpListPair(v2, TypeExpListNil()) :
      Equate(Ult(v1, s), Ult(v2, s), s),
            TypeExpListPair(v2, rest2) :
      UnifyList(rest1, rest2, Equate(Ult(v1, s), Ult(v2, s), s)),
            default : s
      ),
      default : s
    )
};


/* returns length of a list */
INT Length(TYPEEXPLIST l) {
   with(l) (
      TypeExpListPair(v, rest) : 1 + Length(rest),
      default : 0
   )
};


SUBST Equate(TYPEEXP t, TYPEEXP u, SUBST s) {
    t == u ? s :
      with (t) (
        UniversalType(): s,
        TypeVar(v) :
          with(u) (
            UniversalType(): s,
            default: TypeVarOccurCheck(v, u, s) ? FailSubst :
                     SubstConcat(v, u, s),
          ),
        RefType(t1):
          with(u) (
            UniversalType(): s,
            TypeVar(u1): Equate(u, t, s),
            RefType(u1): Unify(t1, u1, s),
            default: FailSubst,
        ),
        MapType(t1, t2):
            with(u) (
               UniversalType(): s,
               TypeVar(u1): Equate(u, t, s),
               MapType(u1, u2): (Length(t1) == Length(u1)) ?
                     Unify(t2, u2, UnifyList(t1, u1, s)) : FailSubst,
               default: FailSubst,
               ),
        PairType(t1, t2):
            with(u) (
               UniversalType(): s,
               TypeVar(u1): Equate(u, t, s),
               PairType(u1, u2): Unify(t2, u2, Unify(t1, u1, s)),
               default: FailSubst,
               ),
        default:
            with(u) (
               UniversalType(): s,
               TypeVar(u1): SubstConcat(u1, t, s),
               default: FailSubst,
               ),
        )
};


BOOL TypeVarOccurCheck(TYPEVAR v, TYPEEXP t, SUBST sub) {
    with(t) (
       UniversalType(): false,
       TypeVar(u): (u == v) || (InSubst(u, sub) &&
                   TypeVarOccurCheck(v, LookupInSubst(u, sub), sub)),
       MapType(t1, t2): TypeVarOccurCheckList(v, t1, sub) ||
                        TypeVarOccurCheck(v, t2, sub),
       PairType(t1, t2): TypeVarOccurCheck(v, t1, sub) ||
                          TypeVarOccurCheck(v, t2, sub),
       RefType(t1): TypeVarOccurCheck(v, t1, sub),
       default: false
       )
};


/* implement TypeVarOccurCheck for a list of  type expressions */
BOOL TypeVarOccurCheckList(TYPEVAR v, TYPEEXPLIST t, SUBST sub) {
    with(t) (
       TypeExpListPair(u, rest) :
                 TypeVarOccurCheck(v, u, sub) ? true
                                 :TypeVarOccurCheckList(v, rest, sub),

       default : false
    )
};


/* Is e a value of Poly C ? */
BOOL NonExpansive (exp e) {
   with(e) (
   Pair(s, t)      : NonExpansive(s) && NonExpansive(t),
   Ident(*)        : true,
   Lambda(*, *)    : true,
   default         : false
   )
};

/* Initial type environment is empty */
TYPEENV InitialEnvironment() { NullTypeEnv };



/*********************************************************************
* File Name : int.ssl                                               *
* Purpose   : Integer operators                                     *
```

```
******************************************************************/

/* Abstract syntax ------------------------------------------------ */
exp :   IntOp(INT)
     |  Sum, Diff, Prod, Quot(exp exp)
     |  LessThan, LessThanOrEqual, GreaterThan,
        GreaterThanOrEqual(exp exp)
     ;

/* Minimal parenthesization --------------------------------------- */
exp :   Sum, Diff  PP2(6)
     |  Prod, Quot PP2(7)
     |  LessThan, LessThanOrEqual, GreaterThan, GreaterThanOrEqual
        PP2(5)
     ;

/* Unparsing ------------------------------------------------------ */
exp :   IntOp       [ ^ ::= ^ ]
     |  Sum  [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:+%S)
              %o " @ "%S(PUNCTUATION:" rp "%S)%}" ]
     |  Diff [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:-%S)
              %o " @ "%S(PUNCTUATION:" rp "%S)%}" ]
     |  Prod [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:*%S)
              %o " @ "%S(PUNCTUATION:" rp "%S)%}" ]
     |  Quot [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ " %S(OPERATOR:/%S)
              %o " @ "%S(PUNCTUATION:" rp "%S)%}" ]
     |  LessThan  [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ "
                  %S(OPERATOR:<%S)%o " @ "%S(PUNCTUATION:" rp "%S)%}"]
     |  LessThanOrEqual [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ " %S(
                OPERATOR:%<le>%S)%o " @ "%S(PUNCTUATION:" rp "%S)%}"]
     |  GreaterThan  [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ " %S(
                    OPERATOR:>%S)%o " @ "%S(PUNCTUATION:" rp "%S)%}"]
     |  GreaterThanOrEqual  [ ^ ::= "%{%S(PUNCTUATION:" lp "%S)" @ "
              %S(OPERATOR:%<ge>%S)%o " @ "%S(PUNCTUATION:" rp "%S)%}"]
    ;

/* Template commands ---------------------------------------------- */
transform exp
   on "+" <exp> : Sum(<exp>, <exp>),
   on "-" <exp> : Diff(<exp>, <exp>),
   on "*" <exp> : Prod(<exp>, <exp>),
   on "/" <exp> : Quot(<exp>, <exp>),
    on "<" <exp> : LessThan(<exp>, <exp>),
    on "<=" <exp> : LessThanOrEqual(<exp>, <exp>),
    on ">" <exp> : GreaterThan(<exp>, <exp>),
    on ">=" <exp> : GreaterThanOrEqual(<exp>, <exp>)
    ;

/* Concrete input syntax ----------------------------------------- */
Exp ::=   (INTEGER)       { Exp$1.abs = IntOp(STRtoINT(INTEGER)); }
     |  (Exp ’+’ Exp)   { Exp$1.abs = Sum( Exp$2.abs, Exp$3.abs); }
     |  (Exp ’-’ Exp)   { Exp$1.abs = Diff(Exp$2.abs, Exp$3.abs); }
     |  (Exp ’*’ Exp)   { Exp$1.abs = Prod(Exp$2.abs, Exp$3.abs); }
     |  (Exp ’/’ Exp)   { Exp$1.abs = Quot(Exp$2.abs, Exp$3.abs); }
     |  (Exp ’<’ Exp)   { Exp$1.abs = LessThan(Exp$2.abs, Exp$3.abs); }
     |   (Exp LESSEQUAL Exp prec LESSEQUAL)
             { Exp$1.abs = LessThanOrEqual(Exp$2.abs, Exp$3.abs); }
     |  (Exp ’>’ Exp) {Exp$1.abs = GreaterThan(Exp$2.abs, Exp$3.abs); }
     |  (Exp GREATEREQUAL Exp prec GREATEREQUAL)
             { Exp$1.abs = GreaterThanOrEqual(Exp$2.abs, Exp$3.abs); }
     ;


/*********************************************************************
* File Name : int_infer.ssl                                         *
* Purpose   : Type inference for integer operators                  *
*********************************************************************/

exp :   IntOp  {
           exp.typeAssignment = IntType;
           exp.S = exp.s;
           exp.partial = false;
        }
    |   Sum, Diff, Prod, Quot  {
           exp$2.typeEnv = exp$1.typeEnv;
           exp$2.letvars = exp$1.letvars;
           exp$3.letvars = exp$1.letvars;
           exp$2.s = exp$1.s;
           exp$3.s = Unify(exp$2.typeAssignment, IntType, exp$2.S);
           exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s,
                                exp$1.typeEnv);
           exp$1.S = Unify(exp$3.typeAssignment, IntType, exp$3.S);
           exp$1.typeAssignment = IntType;
           exp$1.partial = exp$2.partial || exp$3.partial;
           exp$3.sv = exp$1.sv;
           exp$2.sv = exp$1.sv;
           exp$3.encl = exp$1.encl;
           exp$2.encl = exp$1.encl;
           exp$2.top = false;
           exp$3.top = exp$1.top;
        }
    |   LessThan, LessThanOrEqual, GreaterThan, GreaterThanOrEqual  {
           exp$2.typeEnv = exp$1.typeEnv;
           exp$2.letvars = exp$1.letvars;
           exp$3.letvars = exp$1.letvars;
           exp$2.s = exp$1.s;
           exp$3.s = Unify(exp$2.typeAssignment, IntType, exp$2.S);
           exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s,exp$1.typeEnv);
           exp$1.S = Unify(exp$3.typeAssignment, IntType, exp$3.S);
           exp$1.typeAssignment = IntType;
           exp$1.partial = exp$2.partial || exp$3.partial;
           exp$3.sv = exp$1.sv;
           exp$2.sv = exp$1.sv;
           exp$3.encl = exp$1.encl;
           exp$2.encl = exp$1.encl;
           exp$2.top = false;
           exp$3.top = exp$1.top;
        }
    ;

exp :   Sum, Diff, Prod, Quot { in TypeErrors on (exp$1.S == FailSubst
                     && exp$2.S != FailSubst && exp$3.S != FailSubst);
         }
    |   Sum    [ TypeErrors @ : "Sum%n" ^ ^ ]
    |   Diff    [ TypeErrors @ : "Diff%n" ^ ^ ]
    |   Prod    [ TypeErrors @ : "Prod%n" ^ ^ ]
    |   Quot    [ TypeErrors @ : "Quot%n" ^ ^ ]
    |   LessThan, LessThanOrEqual, GreaterThan, GreaterThanOrEqual  {
           in TypeErrors on (exp$1.S == FailSubst && exp$2.S !=
                            FailSubst && exp$3.S != FailSubst);
        }
   |    LessThan          [ TypeErrors @ : "LessThan%n" ^ ^ ]
   |    LessThanOrEqual      [ TypeErrors @ : "LessThanOrEqual%n" ^ ^ ]
   |    GreaterThan          [ TypeErrors @ : "GreaterThan%n" ^ ^ ]
   |    GreaterThanOrEqual
```

```
                    [ TypeErrors @ : "GreaterThanOrEqual%n" ^ ^ ]



/*********************************************************************
* File Name : lambda.ssl                                            *
* Purpose   :                   *
*********************************************************************/
/* An address is a pair of a segment and an offset. */
# define SEGMENT STR
# define OFFSET  STR

/* Formal parameters of a function is a list of identifiers. */
/* Abstract syntax ----------------------------------------------- */
list formalParamList;
formalParamList :   FormalParamListNil()
                |   FormalParamListPair(Id formalParamList)
                ;

FormalParamList { synthesized formalParamList abs; };


/* Actual parameters of an application is a list of expressions. */
/* Abstract syntax ----------------------------------------------- */

list actualParamList;
actualParamList   :   ActualParamListNil()
                  |   ActualParamListPair(exp actualParamList)
                  ;
ActualParamList { synthesized actualParamList abs; };

/*---------------------------------------- */



/* Abstract syntax ----------------------------------------------- */
exp   :   VoidExp()
      |   Refloc, Varloc(LOCATION)
      |   Ident(Id)
      |   Lambda(formalParamList exp)
      |   Call(exp actualParamList)
```

```
    ;

LOCATION :  NullLoc()      [@ : ]
         |  Loc(SEGMENT OFFSET)   [ ^ : "@("^","^")" ]
         ;

/* Minimal parenthesization ---------------------------------------- */
exp { inherited INT precedence; };

# define PP1(n) {\
   local STR lp;\
   local STR rp;\
   exp$2.precedence = (n);\
   lp = ($$.precedence > (n)) ? "(" : "";\
   rp = ($$.precedence > (n)) ? ")" : "";\
   }

# define PP2(n) {\
   local STR lp;\
   local STR rp;\
   exp$2.precedence = (n);\
   exp$3.precedence = (n)+1;\
   lp = ($$.precedence > (n)) ? "(" : "";\
   rp = ($$.precedence > (n)) ? ")" : "";\
   }

/*
 * Values are a subset of the expressions, so SSL expects values to
 * to be attributed as well since expressions are attributed. But the
 * attribution is not important so we define two macros to silence SSL
 */

# define SYNSILENCE(P) P.typeAssignment = NullType;\
   P.S = IdSubst();\
   P.partial = false;

# define INHSILENCE(P) P.typeEnv = NullTypeEnv;\
   P.letvars = LVNil();\
   P.s = IdSubst();\
   P.precedence = 0;\
   P.sv = SVNil();\
   P.encl = true;\
   P.top  = false;


exp   :   Call   PP1(0)
      |   Lambda  PP1(0)
      ;

/* Unparsing ----------------------------------------------------- */
exp :   VoidExp    [ ^ ::= "%S(PLACEHOLDER:<exp>%S)" ]
    |   Ident      [ ^ ::= ^ ]
    |   Refloc     [ ^ : "^"^ ]
    |   Varloc     [ ^ : ^ ]
    |   Call       [ ^ ::= @ "%{%S(PUNCTUATION:(%S)%o" @
                   "%S(PUNCTUATION:)%S)%}"]
    |   Lambda
            [^ ::= "%{%S(PUNCTUATION:" lp "%S)%S(PUNCTUATION:
             %<lambda>(%S)" @ "%S(PUNCTUATION:)%S)%S(PUNCTUATION:
             {%S)%L"  @ "%S(PUNCTUATION: } %S)%S(PUNCTUATION:
             " rp "%S)" "%b%}"]
    ;

/* Template rules ------------------------------------------------ */
transform exp
    on "fun"       e: Lambda(<formalParamList>, e),
    on "call"    <exp> : Call(<exp>, <actualParamList>),
    on "call" e    : Call(e, <actualParamList>)
    ;

/* Concrete input syntax ----------------------------------------- */
Exp { synthesized exp abs; };
exp ~ Exp.abs;
Exp ::=   (EXP_PLACEHOLDER) { Exp.abs = VoidExp; }

    |   (id)   { Exp.abs = Ident(id.abs); }

    |   (LAMBDA '(' FormalParamList ')' '{' Exp '}')
         { Exp$1.abs = Lambda(FormalParamList.abs, Exp$2.abs); }

    |   ('(' Exp ')')
         { Exp$1.abs = Exp$2.abs ; }

    |   (Exp '(' ActualParamList ')' )
```

```
         { Exp$1.abs = Call(Exp$2.abs, ActualParamList.abs); }
    ;


/* Unparsing ----------------------------------------------------- */
formalParamList :  FormalParamListNil   [@:]
                |  FormalParamListPair  [ @ : "%{" ^ ["%S(PUNCTUATION:
                                         ,%S) %o" ] @ "%}"]
                ;

/* Concrete input syntax ----------------------------------------- */
formalParamList ~ FormalParamList.abs;
FormalParamList ::=   (id)   { FormalParamList.abs  =
                                     (id.abs :: FormalParamListNil); }
                |   (id ',' FormalParamList) { FormalParamList$1.abs =
                                  (id.abs :: FormalParamList$2.abs); }
                ;

/* Unparsing ----------------------------------------------------- */
actualParamList :   ActualParamListNil [@:]
                |   ActualParamListPair [ @ : ^ ["%S(PUNCTUATION:,%S)
                                                 %o" ] @]
                ;

/* Concrete input syntax ----------------------------------------- */
actualParamList ~ ActualParamList.abs;
ActualParamList ::=  (Exp)   { ActualParamList.abs  = Exp.abs ::
                                               ActualParamListNil(); }
                |  (Exp ',' ActualParamList) { ActualParamList$1.abs =
                                   Exp.abs :: ActualParamList$2.abs; }
                ;


/********************************************************************
* File Name : lambda_infer.ssl                                       *
* Purpose   :                  *
********************************************************************/

/* Common attributes of exp and actualParamList.
 * Attibutes encl, top and sv are used in checking if
 * the free identifiers of a lambda abstraction are top level.
   encl shows if an expression is enclosed by a lambda abstraction;
   top shows if an expression occurs in a top level scope. For
   instance in letvar x = e_1 in e_2, e_1.top is always false.
   If this letvar expression is enclosed by an expression e then
   e_2.top gets the same value as the value of e.top. Otherwise,
   e_2.top is true; sv is a list of top level identifiers.
 */

exp, actualParamList {
    inherited TYPEENV typeEnv;
    inherited LVLIST letvars;
    synthesized BOOL partial;
    synthesized SUBST S;
    inherited SUBST s;
    inherited BOOL encl;
    inherited BOOL top;
    inherited SVLIST sv;
    };

/* Types of expressions of an actualParamList are hold in
 * texlist. texlist is a TYPEEXPLIST which is implemented
 * using SSL list.
 */
actualParamList { synthesized TYPEEXPLIST texlist; };
exp { synthesized TYPEEXP typeAssignment; };

actualParamList :  ActualParamListPair {
         actualParamList$1.texlist = exp.typeAssignment::
                                         actualParamList$2.texlist;
         exp.typeEnv = actualParamList$1.typeEnv;
         actualParamList$2.typeEnv = ApplySubstToTypeEnv(exp.S,
                                   actualParamList$1.typeEnv);
         exp.letvars = actualParamList$1.letvars;
         actualParamList$2.letvars = actualParamList$1.letvars;
         exp.s = actualParamList$1.s;
         actualParamList$2.s =  exp.S;
         exp.encl = actualParamList$1.encl;
         actualParamList$2.encl = actualParamList$1.encl;
         exp.top = false;
         actualParamList$2.top = false;
         exp.sv = actualParamList$1.sv;
         actualParamList$2.sv = actualParamList$1.sv;
         actualParamList$1.S = actualParamList$2.S;
         actualParamList$1.partial = exp.partial ||
                                             actualParamList$2.partial;
         exp.precedence = 0;
         }
                |  ActualParamListNil {
         actualParamList.texlist = TypeExpListNil;
         actualParamList.S = actualParamList.s;
         actualParamList.partial = false;
};



term :  Static, Dynamic {
      local SUBST finalSubst;
      finalSubst = exp.S;
      exp.typeEnv = InitialEnvironment();
      exp.s = IdSubst;
      exp.letvars = IdNull() :: LVNil;
      local TYPESCHEME finalTypeScheme;
      finalTypeScheme =
         NonExpansive(exp) ? Close(NullTypeEnv,
            RecReal(exp.typeAssignment, exp.S))
            :  TypeExp(RecReal(exp.typeAssignment, exp.S));
      exp.top = true;
      exp.encl = false;
      exp.sv = SVNil();
      }
         ;

term :  Static [ ^ : @ "%n%S(PUNCTUATION::%S) " finalTypeScheme ]
     |  Dynamic {
           local exp val;
           val = (exp.S == FailSubst) || (exp.partial) ?
                Ident(Identifier("?"))
                : let EvalPair(v, *) = eval(exp, NullMem) in (v);
        }
        [ ^ : @ "%nval " val " %S(PUNCTUATION::%S) " finalTypeScheme ]
     ;

exp :  VoidExp {
           exp.typeAssignment = TypeVar(WeakVar(newsymi()));
           exp.S = exp.s;
```

```
       exp.partial = true;
   }

|  Refloc, Varloc {SYNSILENCE(exp)}
|  Ident {
          local TYPESCHEME binding;
          binding = LookupInTypeEnv(Id.name, exp.typeEnv);
          exp.typeAssignment = InstScheme(binding);
          exp.S = binding == TypeExp(UniversalType) ?
               FailSubst /* Free variables cause inconsistency */
               : exp.s;
          exp.partial = Id.partial;
     }

| Call {
     local TYPEVAR beta;
     exp$2.typeEnv = exp$1.typeEnv;
     exp$2.s = exp$1.s;
     exp$2.letvars = exp$1.letvars;
     actualParamList.letvars = exp$1.letvars;
     actualParamList.s = exp$2.S;
     actualParamList.typeEnv = ApplySubstToTypeEnv(exp$2.S,
                               exp$1.typeEnv);
     exp$1.S = Unify(exp$2.typeAssignment,
               MapType(actualParamList.texlist, TypeVar(beta)),
               actualParamList.S);
     beta = WeakVar(newsymi());
     exp$1.typeAssignment = TypeVar(beta);
     exp$1.partial = exp$2.partial || actualParamList.partial;
     actualParamList.sv = exp$1.sv;
     exp$2.sv = exp$1.sv;
     actualParamList.encl = exp$1.encl;
     exp$2.encl = exp$1.encl;
     actualParamList.top = exp$1.top;
     exp$2.top = false;
     }
| Lambda {
     local TYPEEXPLIST formalParamType;
     local TYPEEXP tau;
     formalParamType = GenerateTypeVars(formalParamList);
     exp$1.typeAssignment = tau;
     tau =  Closed(FreeVarsIn(exp$1, BVNil), exp$1.sv) ?
```

```
                       MapType(formalParamType, exp$2.typeAssignment)
              : NullType();

         exp$1.S = ((tau == NullType()) ||
            MultipleOccurrenceIn(formalParamList)) ? FailSubst()
                                                   : exp$2.S;
         exp$2.s = exp$1.s;
         exp$2.letvars = RemoveFPFromLVList(formalParamList,
                                            exp$1.letvars);
         exp$2.typeEnv = TypeEnvConcatList(formalParamList,
                         formalParamType, RemoveFPFromTypeEnv(
                         formalParamList, exp$1.typeEnv));
         exp$1.partial = exp$2.partial;
         exp$2.sv = RemoveFPFromSVList(formalParamList, exp$1.sv);
         exp$2.top = false;
         exp$2.encl = true;
    };



sparse view TypeErrors;

exp :  Ident   { in TypeErrors on (exp.S == FailSubst); }
         [ TypeErrors @ : "Id: " ^ "%n" ]
    |  Lambda   { in TypeErrors on (exp$1.S == FailSubst &&
                  exp$2.S != FailSubst); }
         [ TypeErrors @ : "Lambda%n" ^ ^ ]
    ;

transform term
     on "eval-on"
            Static(e)
               when ((!e.partial) && (e.S != FailSubst)) : Dynamic(e),
     on "eval-off"
            Dynamic(e) : Static(e)
     ;

/* Return the free variables of e wrt bound variables list l */
VLIST FreeVarsIn (exp e, VLIST l) {
  with (e) (
    Ident(Identifier(x)) : InVList(Identifier(x), l)? BVNil
```

```
                                        : BVCons(Identifier(x), BVNil),
    AddrOf(e)         : FreeVarsIn (e,l),
    Subscript(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l) ,
    Assign(e1, e2)    : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l) ,
    PtrAdd(e1, e2)    : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l) ,
    Deref(e)          : FreeVarsIn (e,l),
    Lambda(f, e1) : FreeVarsIn (e1, ConcatFormalParams(f, l)),
    Let(Identifier(x), e1, e2) : FreeVarsIn (e1,l) @
                                 FreeVarsIn (e2,Identifier(x)::l) ,
    LetVar(Identifier(x), e1, e2) : FreeVarsIn (e1,l) @
                                    FreeVarsIn (e2,Identifier(x)::l) ,
    LetArr(Identifier(x),e1,e2) : FreeVarsIn (e1,l) @
                                  FreeVarsIn (e2,Identifier(x)::l) ,
    Compose(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Not(e1) : FreeVarsIn (e,l),
    And(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Or(e1, e2)  : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Equal(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    NotEqual(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Cond(e1, e2, e3) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l) @
                       FreeVarsIn (e3,l),
    While(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Sum(e1, e2)   : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Diff(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Prod(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    Quot(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn (e2,l),
    LessThan(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn(e2,l),
    LessThanOrEqual(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn(e2,l),
    GreaterThan(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn(e2,l),
    GreaterThanOrEqual(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn(e2,l),
    Pair(e1, e2) : FreeVarsIn (e1,l) @ FreeVarsIn(e2,l),
    Call(e,a)    : FreeVarsIn (e,l) @ FreeVarsInList(a,l),
    default: BVNil() /* constants and placeholders */
  )
};

VLIST ConcatFormalParams(formalParamList l, VLIST bv) {
   with(l) (
     FormalParamListPair(v, rest) : ConcatFormalParams(rest,v :: bv),
     FormalParamListNil : bv
   )
};
```

```
/* A more general form of FreeVarsIn for finding the
 * free variables in a list of expressions.
 */
VLIST FreeVarsInList(actualParamList l, VLIST bv) {
   with(l) (
     ActualParamListPair(e,rest) :
        FreeVarsIn(e, bv) @ FreeVarsInList(rest, bv),
     default : BVNil()
   )
};

/* Is fv a subset of l. In other words, we check
 * if all the free varibles given by fv occur in l
 */
BOOL Closed(VLIST fv, SVLIST l) {
   with(fv) (
      BVNil : true,
      BVCons(v, rest) :
         InSVList(v, l) ? Closed(rest, l)
         : false
   )
};

BOOL InSVList (Id id, SVLIST l) {
   with (l) (
      SVNil : false,
      SVCons (v, rest) : (v == id) ? true
               : InSVList(id, rest)
   )
};



SVLIST RemoveFromSVList (Id id, SVLIST l) {
   with (l) (
      SVNil : l,
      SVCons(v, rest) : (v == id) ? rest :
         v :: RemoveFromSVList(id, rest)
   )
};
```

```
?* Remove let/letvar/letarr bound variables given
 * l from b.
 */
VLIST RemoveLetbounds (VLIST b, LVLIST l) {
   with (b) (
      BVNil() : BVNil(),
      BVCons(v, rest) :
         InLVList(v, l) ? v::RemoveLetbounds (rest,l)
                : RemoveLetbounds (rest,l)
      )
};


/* Generate new type variables for the formal parameters
 * of a function.
 */
TYPEEXPLIST GenerateTypeVars(formalParamList l) {
   with (l) (
      FormalParamListPair(f, rest) :
         TypeExpListPair(TypeVar(WeakVar(newsymi())),
                         GenerateTypeVars(rest)),
      default : TypeExpListNil()
   )
};

/* Remove the formal parameters from type environment */
TYPEENV RemoveFPFromTypeEnv(formalParamList l, TYPEENV t ) {
   with(l) (
      FormalParamListPair(Identifier(id), rest) :
         RemoveFromTypeEnv(id,RemoveFPFromTypeEnv(rest, t)),
      default : t
   )
};


/* Add type assumptions for the formal parameters given by l
 * to the type environment. Each formal parameter f in
 * position x of l, is associated with the type expression given in
 * position x of type expression list e.
 */
TYPEENV TypeEnvConcatList(formalParamList l, TYPEEXPLIST e,TYPEENV t){
    with(l) (
      FormalParamListPair(Identifier(id), rest1) :
          with (e) (
            TypeExpListPair(v, rest2) : TypeEnvConcat(id, TypeExp(v),
          TypeEnvConcatList(rest1, rest2, t)),
          default : t
          ),
      default : t
    )
};


LVLIST RemoveFPFromLVList(formalParamList l, LVLIST lv) {
   with(l) (
     FormalParamListPair(v,rest) :
       RemoveFromLVList(v,RemoveFPFromLVList(rest, lv)),
     default : lv
   )
};

SVLIST RemoveFPFromSVList(formalParamList l, SVLIST sv) {
   with(l) (
     FormalParamListPair(v,rest) :
       RemoveFromSVList(v,RemoveFPFromSVList(rest, sv)),
     default : sv
   )
};


/* Functions can only have distinct formal parameters. */
BOOL MultipleOccurrenceIn(formalParamList l) {
   with(l) (
      FormalParamListNil : false,
      FormalParamListPair(x, rest) :
         Occur(x, rest) ? true :MultipleOccurrenceIn(rest)
   )
};

BOOL Occur(Id x, formalParamList l) {
   with(l) (
      FormalParamListNil : false,
      FormalParamListPair(y, rest) :
         (x == y) ? true : Occur(x, rest)
   )
};

/********************************************************************
* File Name : let.ssl                                               *
* Purpose   : let and letvar declarations                           *
********************************************************************/

/* Abstract syntax ---------------------------------------------- */
exp :   Let(Id exp exp)
    |   LetVar(Id exp exp)
    ;

/* Minimal parenthesization ------------------------------------- */
exp :   Let, LetVar {
            exp$2.precedence = 0;
            exp$3.precedence = 0;
        }
    ;

/* Unparsing ---------------------------------------------------- */
exp :  Let [ ^ ::= "%{%L%S(KEYWORD:let%S) " @ " = " @
           " %S(KEYWORD:in%S)%t%t%n" @ "%b%b%n%S(KEYWORD:end%S)%b%}" ]
   |   LetVar  [ ^ ::= "%{%L%S(KEYWORD:letvar%S) " @ " := " @
           " %S(KEYWORD:in%S)%t%t%n" @ "%b%b%n%S(KEYWORD:end%S)%b%}" ]
   ;

/* Template commands -------------------------------------------- */
transform exp
   on "let" <exp>: Let(<Id>, <exp>, <exp>),
   on "let<Id><exp>e" e when (e != <exp>): Let(<Id>, <exp>, e),
   on "let<Id>e<exp>" e when (e != <exp>): Let(<Id>, e, <exp>),
   on "letvar" <exp>: LetVar(<Id>, <exp>, <exp>),
   on "letvar<Id><exp>e" e when (e != <exp>): LetVar(<Id>, <exp>, e),
   on "letvar<Id>e<exp>" e when (e != <exp>): LetVar(<Id>, e, <exp>)
   ;

/* Concrete input syntax ---------------------------------------- */
Exp ::=  (LET id '=' Exp IN Exp END) {
             Exp$1.abs = Let(id.abs, Exp$2.abs, Exp$3.abs);
             }
```

```
  |    (LETVAR id ASSIGN Exp IN Exp END) {
                 Exp$1.abs = LetVar(id.abs, Exp$2.abs, Exp$3.abs);
                 }
 ;


/*****************************************************************
* File Name : let_infer.ssl                                       *
* Purpose   : Type inference for let and letvar                   *
*****************************************************************/

/*
 * Two local attributes, sigma and finalTypeScheme, are needed in the
 * attribution of Let; sigma is used to extend the type environment,
 * while finalTypeScheme gives the typing used in the alternative
 * unparsing rule. Type sigma may not be a final type scheme for
 * Id.name because it may contain type variables that get specialized
 * by an enclosing expression. e.g, letvar x=[] in
 * let y = (let z=x in 17) in 1::x. The type of z is determined by
 * "1::x" of the enclosing expression "let y = ...".
 * Thus the final type scheme must be formed from the final
 * substitution finalSubst inherited from the root.  This is done
 * using the upward remote attribute set {Static.finalSubst,
 * Dynamic.finalSubst}.
 *
 * If attribute finalTypeScheme is used for both purposes, then a
 * type 2 circularity results--there is a mutual dependence between
 * finalTypeScheme and finalSubst.
 *
 * Likewise local attribute tau of LetVar, used in the alternative
 * unparsing rule, must also be formed from finalSubst.
 */

exp :  Let {
         local TYPESCHEME sigma;
         local TYPESCHEME finalTypeScheme;

         exp$1.S = exp$3.S;
         exp$1.typeAssignment = exp$3.typeAssignment;
         exp$1.partial = Id.partial || exp$2.partial || exp$3.partial;
         exp$2.s = exp$1.s;
         exp$2.letvars = exp$1.letvars;
```

```
      exp$3.letvars = RemoveFromLVList(Id, exp$1.letvars);
      exp$2.typeEnv = exp$1.typeEnv;
      exp$3.s = exp$2.S;
      exp$3.typeEnv = TypeEnvConcat(Id.name, sigma,
               ApplySubstToTypeEnv(exp$2.S,
               RemoveFromTypeEnv(Id.name, exp$1.typeEnv)));
      sigma =
        NonExpansive(exp$2) ?
        Close(ApplySubstToTypeEnv(exp$2.S, exp$1.typeEnv),
              RecReal(exp$2.typeAssignment, exp$2.S))
        : TypeExp(RecReal(exp$2.typeAssignment, exp$2.S));
      finalTypeScheme =
        NonExpansive(exp$2) ?
        Close(ApplySubstToTypeEnv({Static.finalSubst,
              Dynamic.finalSubst}, exp$1.typeEnv),
              RecReal(exp$2.typeAssignment, {Static.finalSubst,
              Dynamic.finalSubst}))
        : TypeExp(RecReal(exp$2.typeAssignment,
         {Static.finalSubst,Dynamic.finalSubst}));
      exp$2.sv = exp$1.sv;
      exp$3.sv = exp$1.top ? exp$1.encl ? RemoveFromSVList(Id,
                                          exp$1.sv)
                                        : SVCons(Id,exp$1.sv)
                           : exp$1.sv;
      exp$3.encl = exp$1.encl;
      exp$2.encl = exp$1.encl;
      exp$2.top = false;
      exp$3.top = exp$1.top;
            }
|  LetVar {
      local TYPEEXP tau;

      exp$1.S = exp$3.S;
      exp$1.typeAssignment = exp$3.typeAssignment;
      exp$1.partial = Id.partial || exp$2.partial || exp$3.partial;
      exp$2.s = exp$1.s;
      exp$2.letvars = exp$1.letvars;
      exp$3.letvars = (Id == IdNull()) ? exp$1.letvars
                       : Id :: RemoveFromLVList(Id, exp$1.letvars);
      exp$2.typeEnv = exp$1.typeEnv;
      exp$3.s = FreeInLambda(Id.name, exp$3) ?
                   Unify(TypeVar(WeakVar(newsymi())),
                            exp$2.typeAssignment, exp$2.S)
                      : exp$2.S;
         exp$3.typeEnv =
                 TypeEnvConcat(Id.name, TypeExp(exp$2.typeAssignment),
                  ApplySubstToTypeEnv(exp$3.s, RemoveFromTypeEnv(
                  Id.name, exp$1.typeEnv)));
         /* use RecReal here only because alternative unparsing rule
            displays type tau so type must be closed */
         tau = RecReal(exp$2.typeAssignment, {Static.finalSubst,
                Dynamic.finalSubst});
         exp$2.sv = exp$1.sv;
         exp$3.sv = exp$1.top ? exp$1.encl ? RemoveFromSVList(Id,
                                             exp$1.sv)
                                           : SVCons(Id,exp$1.sv)
                              : exp$1.sv;
         exp$3.encl = exp$1.encl;
         exp$2.encl = exp$1.encl;
         exp$2.top = false;
         exp$3.top = exp$1.top;
    }
   ;

/* Alternative unparsing ------------------------------------- */

exp :   Let  [ ^ ::= "%{%L%S(KEYWORD:let%S) " @ ":" finalTypeScheme
              " = %o" @ " %S(KEYWORD:in%S)%t%t%n" @ "%b%b%n
              %S(KEYWORD:end%S)%b%}" ]
    |   LetVar  [ ^ ::= "%{%L%S(KEYWORD:letvar%S) " @ ":"
                 tau " var := %o" @ " %S(KEYWORD:in%S)%t%t%n" @
                 "%b%b%n%S(KEYWORD:end%b%S)%}" ]
    ;

/* Does id occur free in a \-abstraction in e? */

BOOL FreeInLambda (ID id, exp e) {
 with (e) (
   AddrOf(e)    : FreeInLambda(id, e),
   Subscript(e1, e2) :FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Assign(e1, e2)  : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   PtrAdd(e1, e2)   : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Deref(e)     : FreeInLambda(id, e),
   Lambda(*,*)      : FreeIn(id, e),
   Let(*,e1,e2)     : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   LetVar(*,e1,e2)   : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   LetArr(*,e1,e2)   : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Compose(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Not(e)       : FreeInLambda(id, e),
   And(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Or(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Equal(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   NotEqual(e1,e2) : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Cond(e1,e2,e3)    : FreeInLambda(id, e1) || FreeInLambda(id, e2)
            || FreeInLambda(id, e3),
   While(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Sum(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Diff(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Prod(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Quot(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   LessThan(e1,e2) : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   LessThanOrEqual(e1,e2):
            FreeInLambda(id, e1) || FreeInLambda(id, e2),
   GreaterThan(e1,e2) : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   GreaterThanOrEqual(e1,e2) : FreeInLambda(id, e1) ||
            FreeInLambda(id, e2),
   Pair(e1,e2)    : FreeInLambda(id, e1) || FreeInLambda(id, e2),
   Call(e,l)   : FreeInLambda(id, e)  || FreeInLambdaList(id, l),
   default      : false /* constants and placeholders */
   )
};

BOOL FreeInLambdaList(ID id, actualParamList l) {
   with(l) (
     ActualParamListPair(e, rest) :
      (FreeInLambda(id, e) || FreeInLambdaList(id, rest)),
     default : false
   )
};

BOOL FreeIn (ID id, exp e) { /* Does id occur free in e? */
  with (e) (
   Ident(Identifier(x)) : id == x,
   AddrOf(e)   : FreeIn(id, e),
   Subscript(e1, e2) :FreeIn(id, e1) || FreeIn(id, e2),
   Assign(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    PtrAdd(e1, e2)   : FreeIn(id, e1) || FreeIn(id, e2),
    Deref(e) : FreeIn(id, e),
    Lambda(f, e1) : !OccursIn(id, f) && FreeIn(id, e1),
    Let(Identifier(x), e1, e2) : FreeIn(id, e1) ||
                                 (FreeIn(id, e2) && id != x),
    LetVar(Identifier(x), e1, e2) : FreeIn(id, e1) ||
                                    (FreeIn(id, e2) && id != x),
    LetArr(*,e1,e2)   : FreeIn(id, e1) || FreeIn(id, e2),
    Compose(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Not(e1) : FreeIn(id, e1),
    And(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Or(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Equal(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    NotEqual(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Cond(e1, e2, e3) : FreeIn(id, e1) || FreeIn(id, e2) ||
                       FreeIn(id, e3),
    While(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Sum(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Diff(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Prod(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Quot(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    LessThan(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    LessThanOrEqual(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    GreaterThan(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    GreaterThanOrEqual(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Pair(e1, e2) : FreeIn(id, e1) || FreeIn(id, e2),
    Call(e,l)   : FreeIn(id,e) || FreeInList(id, l),
    default: false /* constants and placeholders */
    )
};

BOOL OccursIn(ID id, formalParamList l) {
   with (l) (
     FormalParamListPair(Identifier(x), rest) :
                   ((x == id) || OccursIn(id, rest)),
     default : false
   )
};

BOOL FreeInList(ID id, actualParamList l) {
   with(l) (
     ActualParamListPair(e, rest) :
             (FreeIn(id, e) || FreeInList(id, rest)),
     default : false
   )
};


/*******************************************************************
* File Name : letarr.ssl                                           *
* Purpose   : Definitions for  letarr, pointer arithmetic and      *
*             array indexing. We make a minor change to Poly C     *
*             syntax and denote pointer arithmetic with special    *
*             character \oplus which is a plus sign + and a circle *
*             around it. But in template panel of the editor this  *
*             sign will be seen as o+ because the current SynGen   *
*             environment can not display this special character   *
*             appropriately.                                       *
*******************************************************************/

/* Abstract syntax -------------------------------------------- */
exp :   LetArr(Id exp exp)
    |   PtrAdd(exp exp)
    |   Subscript(exp exp)
    |   SubscriptL(exp exp) /* For internal use only. */
    ;

/* Minimal parenthesization ----------------------------------- */
exp :  LetArr {
         exp$2.precedence = 0;
         exp$3.precedence = 0;
       }
    |  PtrAdd PP2(6)
    |  Subscript PP2(0)
    ;

/* Unparsing -------------------------------------------------- */
exp :  LetArr [ ^ ::= "%S(KEYWORD:letarr%S) " @  "["@"]"
               " %S(KEYWORD:in%S)%t%t%n" @ "%b%b%n%S(KEYWORD:end%S)" ]
    |  PtrAdd [ ^ ::= "%{" "%S(PUNCTUATION:" lp "%S)" @ "%S(OPERATOR:
               \<oplus>%S)%o " @ "%S(PUNCTUATION:" rp "%S)%}" ]
    |  Subscript     [ ^ ::=  "%{" @ "[" @ "]%}" ]
    |  SubscriptL     [ ^ ::=  "%{" @ "[" @ "]%}" ]
    ;
```

```
/* Template commands --------------------------------------------- */
transform exp
   on "letarr" <exp>: LetArr(<Id>, <exp>, <exp>),
   on "letarr<Id><exp>e" e when (e != <exp>): LetArr(<Id>, <exp>, e),
   on "letarr<Id>e<exp>" e when (e != <exp>): LetArr(<Id>, e, <exp>),
   on "\<oplus>" <exp> : PtrAdd(<exp> , <exp>),
   on "[ ]" <exp> : Subscript(<exp>, <exp>)
   ;

/* Concrete input syntax ----------------------------------------- */
Exp ::= (LETARR id '['Exp']' IN Exp END) {
                    Exp$1.abs = LetArr(id.abs, Exp$2.abs, Exp$3.abs);
                    }
    |   (Exp PTRADD Exp) {Exp$1.abs = PtrAdd( Exp$2.abs, Exp$3.abs); }
    |   (Exp '['Exp']') {Exp$1.abs = Subscript(Exp$2.abs, Exp$3.abs);}
    ;


/**********************************************************************
* File Name : letarr.ssl                                              *
* Purpose   : Type inference for  letarr, pointer arithmetic and      *
*             array indexing.                                         *
**********************************************************************/
exp :  LetArr {
         exp$1.S = exp$3.S;
         exp$1.typeAssignment = exp$3.typeAssignment;
         exp$1.partial = Id.partial || exp$2.partial || exp$3.partial;
         exp$2.s = exp$1.s;
         exp$2.letvars = exp$1.letvars;
         exp$3.letvars = RemoveFromLVList(Id, exp$1.letvars);
         exp$2.typeEnv = exp$1.typeEnv;
         exp$3.s = Unify(exp$2.typeAssignment,IntType,exp$2.S);
         exp$3.typeEnv =
             TypeEnvConcat(Id.name, TypeExp(RefType(TypeVar(
             WeakVar(newsymi())))), ApplySubstToTypeEnv(exp$2.S,
             RemoveFromTypeEnv(Id.name, exp$1.typeEnv)));
         exp$2.sv = exp$1.sv;
         exp$3.sv = exp$1.top ? exp$1.encl ?
                                   RemoveFromSVList(Id, exp$1.sv)
                                   : SVCons(Id,exp$1.sv)
                                : exp$1.sv;
```

```
     exp$3.encl = exp$1.encl;
     exp$2.encl = exp$1.encl;
     exp$2.top = false;
     exp$3.top = exp$1.top;
   }
|  PtrAdd  {
     exp$2.typeEnv = exp$1.typeEnv;
     exp$2.letvars = exp$1.letvars;
     exp$3.letvars = exp$1.letvars;
     exp$2.s = exp$1.s;
     exp$3.s = Unify(RefType(TypeVar(WeakVar(newsymi()))),
               exp$2.typeAssignment, exp$2.S);
     exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s, exp$1.typeEnv);
     exp$1.S = Unify(exp$3.typeAssignment, IntType, exp$3.S);
     exp$1.typeAssignment =
               ApplySubstToTypeExp(exp$1.S, exp$2.typeAssignment);
     exp$1.partial = exp$2.partial || exp$3.partial;
     exp$3.encl = exp$1.encl;
     exp$2.encl = exp$1.encl;
     exp$2.top = false;
     exp$3.top = exp$1.top;
     exp$2.sv = exp$1.sv;
     exp$3.sv = exp$1.sv;
   }
|  Subscript {
     local TYPEEXP tau;
     exp$2.typeEnv = exp$1.typeEnv;
     exp$2.letvars = exp$1.letvars;
     exp$3.letvars = exp$1.letvars;
     exp$2.s = exp$1.s;
     exp$3.s =  Unify(RefType(TypeVar(WeakVar(newsymi()))),
                exp$2.typeAssignment, exp$2.S);
     exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s, exp$1.typeEnv);
     exp$1.S = Unify(exp$3.typeAssignment, IntType, exp$3.S);
     exp$1.typeAssignment =
        with(tau) (
              RefType(t) : t,
              default    : NullType
           );
     exp$1.partial = exp$2.partial || exp$3.partial;
     tau = ApplySubstToTypeExp(exp$1.S, exp$2.typeAssignment);
     exp$3.encl = exp$1.encl;
```

```
          exp$2.encl = exp$1.encl;
          exp$2.top = false;
          exp$3.top = exp$1.top;
          exp$2.sv = exp$1.sv;
          exp$3.sv = exp$1.sv;
        }
    |   SubscriptL {
          INHSILENCE(exp$2) /* this attribution is a result */
          INHSILENCE(exp$3) /* of values being expressions  */
          SYNSILENCE(exp$1)
        }
    ;


/* Alternative unparsing ------------------------------------ */

exp :  PtrAdd {
          in TypeErrors on (exp$1.S == FailSubst &&
             exp$2.S != FailSubst && exp$3.S != FailSubst);
       }    [ TypeErrors @ : "PtrAdd%n" ^ ^ ]
     | Subscript {
          in TypeErrors on (exp$1.S == FailSubst &&
                            exp$2.S != FailSubst);
       }    [ TypeErrors @ : "Subscript%n" ^ ^ ]
     ;


/****************************************************************
* File Name : lex.ssl                                           *
* Purpose   : Lexical syntax, token precedences for concrete input  *
*             syntax and style declarations.                    *
****************************************************************/

/* Lexical syntax --------------------------------------------- */
WHITESPACE  :     WhiteSpaceLex   < [\ \t\n] >;
EXP_PLACEHOLDER:   ExpPlaceholderLex < "<exp>" >;
IDENTIFIER_PLACEHOLDER:   IdentifierPlaceholderLex < "<identifier>" >;
LAMBDA       :   LambdaLex   < "lambda"|"LAMBDA"|{lambda} >;
VAL          :   ValLex      < "val"|"VAL" >;
FIX          :   FixLex      < "fix" >;
```

```
LET         :  LetLex      < "let" >;
LETVAR      :  LetVarLex   < "letvar" >;
LETARR      :  LetArrLex       < "letarr" >;
IN          :  InLex      < "in"|"IN" >;
NIL         :  NilLex      < "nil"|"[]" >;
IF          :  IfLex      < "if"|"IF" >;
WHILE       :  WhileLex   < "while"|"WHILE" >;
UNIT        :  UnitLex      < "unit" >;
THEN        :  ThenLex      < "then"|"THEN" >;
ELSE        :  ElseLex      < "else"|"ELSE" >;
DO          :  DoLex      < "do"|"DO" >;
OD          :  OdLex      < "od"|"OD" >;
FI          :  FiLex      < "fi"|"FI" >;
BEGIN       :  BeginLex   < "begin"|"BEGIN" >;
END         :  EndLex      < "end"|"END" >;
TRUE        :  TrueLex      < "true"|"TRUE" >;
FALSE       :  FalseLex   < "false"|"FALSE" >;
ASSIGN      :  AssignLex   < ":=" >;
LOGICALAND  :  LogicalAnd   < "&&" >;
LOGICALOR   :  LogicalOr   < "||" >;
NOTEQUAL    :  NotEqualLex   < "<>"|{ne} >;
LESSEQUAL   :  LessEqualLex   < "<="|{le} >;
GREATEREQUAL:  GreaterEqualLex   < ">="|{ge} >;
INTEGER     :  IntegerLex   < \-?[0-9]+ >;
FLOAT       :  FloatLex   <[0-9]*(\.[0-9]*)([dDeE][-+]?[0-9]+)?>;
ID          :  IdLex      < [A-Za-z][0-9A-Za-z_$]*[']*|[?] >;
PTRADD      :  PtrAddLex       < {oplus} >;


/* Token precedences for concrete input syntax --------------*/
left LOGICALOR;
left LOGICALAND;
nonassoc NOTEQUAL;
nonassoc '=', '<', LESSEQUAL, '>', GREATEREQUAL;
left PTRADD, '+', '-';
left '*', '/';
right '&', '!', '~';
nonassoc ID, VAL, FIX, IN, NIL, TRUE, FALSE, FLOAT, INTEGER, LET,
         LETVAR,LETARR, IF, WHILE, UNIT, THEN, ELSE, DO, OD, FI,
         BEGIN, END, ASSIGN, LAMBDA, EXP_PLACEHOLDER ;
```

```
/* Style declarations -------------------------------------------- */
style NORMAL, KEYWORD, PLACEHOLDER, PUNCTUATION, OPERATOR;




/********************************************************************
* File Name : newsymi.c                                             *
* Purpose   : New type variable generator.                          *
********************************************************************/
/* $Revision: 1.2 $
 * $Date: 1993/09/02 21:21:12 $
 * $Author: volpano $
 * $Log: newsymi.c,v $
 * Revision 1.2  1993/09/02  21:21:12  volpano
 * Removed T in sprintf.
 *
 */
/*
 *    Copyright (c) 1989, an unpublished work by GrammaTech, Inc.
 *                    ALL RIGHTS RESERVED
 *
 *    This software is furnished under a license and may be used and
 *    copied only in accordance with the terms of such license and the
 *    inclusion of the above copyright notice.  This software or any
 *    other copies thereof may not be provided or otherwise made
 *    available to any other person.  Title to and ownership of the
 *    software is retained by GrammaTech, Inc.
 */

#include "str0_exp.h"
#include "structures_exp.h"
#include "types_exp.h"

/*
 *  newsymi
 *
 *  Generate new unique symbol.
 *
 *  WARNING:  In general, this is not a good technique, because
 *  gratuitous new symbols will cause AFFECTED to be too large.
 */
```

```
FOREIGN newsymi()
{
     static int i;
     static char buff[10];

     sprintf(buff,"*%d",i++);
     return(Str(str_to_str0(buff)));
}
```

```
/********************************************************************
* File Name : pair.ssl                                              *
* Purpose   : Defitions for pair. Pair is the stdout of the         *
*             interpreter. We output the result produced by a       *
*             program through pair construct. One might consider    *
*             using list construct for this purpose. But a list     *
*             requires the elements have the same which is a severe *
*             restriction. Notice that we define only the required  *
*             constructor and do not define first and second        *
*             operations since pair is not in Poly C calculus they  *
*             are not needed.                                       *
********************************************************************/

/* Abstract syntax ----------------------------------------------- */
exp :  Pair(exp exp)
    ;

/* Minimal parenthesization -------------------------------------- */
exp :  Pair {
         exp$2.precedence = 0;
         exp$3.precedence = 0;
       }
    ;

/* Unparsing ----------------------------------------------------- */
exp :   Pair   [ ^ ::= "%S(PUNCTUATION:(%S)" @
                "%S(PUNCTUATION:,%S) %o"  @ "%S(PUNCTUATION:)%S)" ]
    ;

/* Template commands --------------------------------------------- */
transform exp
```

```
    on "( , )" <exp> : Pair(<exp>,<exp>)
    ;

/* Concrete input syntax --------------------------------------- */
Exp ::=  ('(' Exp ',' Exp ')') {$$.abs = Pair(Exp$2.abs, Exp$3.abs);}
    ;




/*****************************************************************
* File Name : pair_infer.ssl                                     *
* Purpose   : Type inference for pair.                           *
*****************************************************************/

exp :  Pair     {
        exp$2.typeEnv = exp$1.typeEnv;
        exp$2.letvars = exp$1.letvars;
        exp$3.letvars = exp$1.letvars;
        exp$2.s = exp$1.s;
        exp$3.typeEnv = ApplySubstToTypeEnv(exp$2.S, exp$1.typeEnv);
        exp$3.s = exp$2.S;
        exp$1.partial = exp$2.partial || exp$3.partial;
        exp$1.S = exp$3.S;
        exp$1.typeAssignment = PairType(exp$2.typeAssignment,
                                        exp$3.typeAssignment);
        exp$3.top = false;
        exp$2.top = false;
        exp$3.encl = exp$1.encl;
        exp$2.encl = exp$1.encl;
        exp$3.sv = exp$1.sv;
        exp$2.sv = exp$1.sv;
        }
    ;


/*****************************************************************
* File Name : real.ssl                                           *
* Purpose   : Definitions for real numbers.                      *
*****************************************************************/

/* Abstract syntax --------------------------------------------- */
```

```
exp :  RealOp(REAL)   [ ^ ::= ^ ]
    ;

/* Concrete input syntax ----------------------------------------- */
Exp ::=  (FLOAT)          { Exp$1.abs = RealOp(STRtoREAL(FLOAT)); }
    ;


/********************************************************************
* File Name : real_infer.ssl                                        *
* Purpose   : Type inference for real numbers.                      *
********************************************************************/

exp :  RealOp  {
         exp.typeAssignment = RealType;
         exp.S = exp.s;
         exp.partial = false;
       }
    ;

/********************************************************************
* File Name : while.ssl                                             *
* Purpose   : Definitions for while loop.                           *
********************************************************************/

/* Abstract syntax ----------------------------------------------- */
exp :  While(exp exp);

/* Minimal parenthesization -------------------------------------- */
exp :  While {
         exp$2.precedence = 0;
         exp$3.precedence = 0;
       }
    ;

/* Unparsing ----------------------------------------------------- */
exp :  While [^ ::= "%t%S(KEYWORD:while%S) " @ " %S(KEYWORD:do%S)\n"
              @ "%b%n%S(KEYWORD:od%S)"]
    ;

/* Template commands --------------------------------------------- */
transform exp
```

```
   on "while" e : While(<exp>, e)
   ;

/* Concrete input syntax ----------------------------------------- */
Exp ::=   (WHILE Exp DO Exp OD)
             { Exp$1.abs = While(Exp$2.abs, Exp$3.abs); }
    ;


/********************************************************************
* File Name : while_infer.ssl                                       *
* Purpose   : Type inference for while loop.                        *
********************************************************************/

/* type inference */
exp :  While {
          exp$2.typeEnv = exp$1.typeEnv;
          exp$2.letvars = exp$1.letvars;
          exp$3.letvars = exp$1.letvars;
          exp$2.s = exp$1.s;
          exp$3.s = Unify(exp$2.typeAssignment, IntType, exp$2.S);
          exp$3.typeEnv = ApplySubstToTypeEnv(exp$3.s, exp$1.typeEnv);
          exp$1.S = exp$3.S;
          exp$1.typeAssignment = UnitType;
          exp$1.partial = exp$2.partial || exp$3.partial;
          exp$3.encl = exp$1.encl;
          exp$2.encl = exp$1.encl;
          exp$3.sv = exp$1.sv;
          exp$2.sv = exp$1.sv;
          exp$2.top = false;
          exp$3.top = false;
       }
    ;

exp :  While
          { in TypeErrors on (exp$3.s == FailSubst &&
                              exp$2.S != FailSubst); }
             [ TypeErrors @ : "While%n" ^ ^ ]
    ;
```

LIST OF REFERENCES

[SmV96a] Geoffrey Smith and Dennis Volpano. Towards an ML-style type system for C. *European Symposium on Programming Systems, Linköping Sweden, LNCS 1058:341-355*, 1996.

[SmV96b] Geoffrey Smith and Dennis Volpano. Polymorphic typing of variables and references. *ACM Trans on Programming Languages and Systems*, 18:3, May 1996.

[DaM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. *ACM Symposium on Principles of Programming Languages*, pages 207-212, 1982.

[Dam85] Luis M. M. Damas. *Type Assignment in Programming Languages.* Ph.D. Thesis, University of Edinburgh, 1985.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Computation, 89*, 12:1–34, 1990.

[Hin69] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transaction of the American Mathematical Society*, 146:29–60, 1969.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer Systems and Sciences*, 17:348–375, 1978.

[Rob65] Julia A. Robinson. A machine-oriented logic based on the resolution priciple. *Journal of ACM*, 12:23–41, 1965.

[KR78] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Gram] GrammaTech Inc. *The Synthesizer Generator Reference Manual.* GrammaTech, Inc., One Hopkins Place, Ithaca, NY. (Fourth Edition), 1993.

[Cor90] Cormen et.al. *The Algorithms.* Prentice-Hall, 1990.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[WrF91] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

[Pfe96] Frank Pfenning. The practice of logical frameworks. *Trees in Algebra and Programming, Linköping Sweden, LNCS 1059:119-134*, 1996.

[Ohor95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transaction on Programming Languages and Systems, 17:844-895*, 1995.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. Chairman, Code CS/LT 2
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5118

4. Dennis Volpano, Code CS/VO 8
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5118

5. Geoffrey Smith, 1
   School of Computer Science
   Florida Int'l University
   University Park
   Miami, Florida 33199

6. Craig Rasmussen, Code MA/RA 1
   Mathematics Department
   Naval Postgraduate School
   Monterey, CA 93943-5101

7. Mustafa Özgen 2
   Gazi Mahallesi Celal Bayar Cad. Ögretmenler Apt.
   No: 14 33960-Silifke ICEL/TURKEY

8. Valdis Berzins, Code CS/BE 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5118